



CAMBRIDGE

Lecture 2: Model Fitting and Optimization

Stefan Bucher

MACHINE LEARNING IN ECONOMICS
UNIVERSITY OF CAMBRIDGE

Stefan Bucher



Prince (2023, chaps. 2, 6, 8).¹

- . Figures taken or adapted from Prince (2023). All rights belong to the original author and publisher. These materials are intended solely for educational purposes.

What is a Model?

“A model is a probability distribution over a sequence of [observable] random variables.” — Thomas Sargent

Review: Linear Regression

What did we do?

- We chose a model (distribution) $\mathbf{Y}|\mathbf{X} \sim \mathcal{N}(\mathbf{X}\beta, \sigma^2 \mathbf{I}_n)$ with parameters β
- We chose an objective function (loss function)
$$L(\beta) = \frac{1}{n} \sum_{i=1}^n (y_i - X_i \beta)^2$$
- We (analytically) found the optimal parameter values $\hat{\beta}$ that minimize the loss function

$$\hat{\beta} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}$$

Model Fitting & Optimization

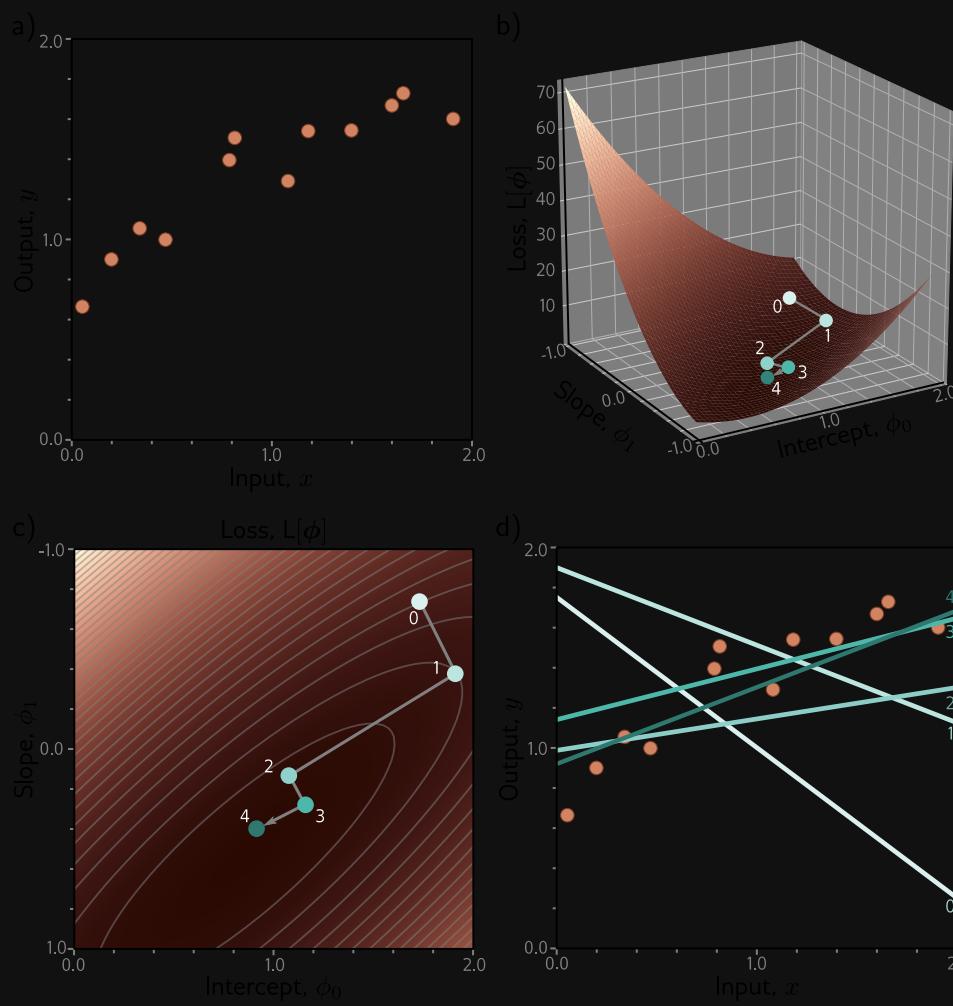
$$\hat{\phi} = \arg \min_{\phi} L(\phi)$$

Prince (2023, chap. 6)

Gradient Descent¹

$$\phi \leftarrow \phi - \alpha \nabla_{\phi} L(\phi)$$

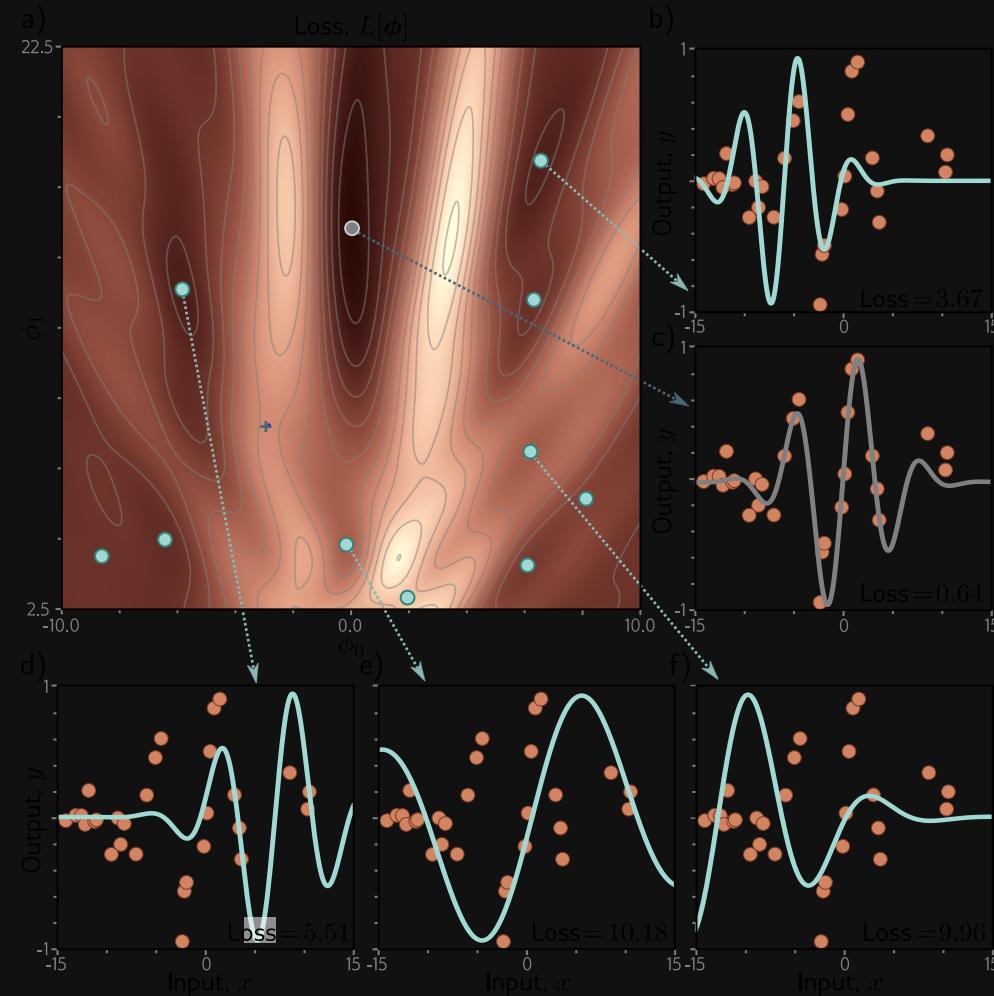
where α fixed learning rate or line search.



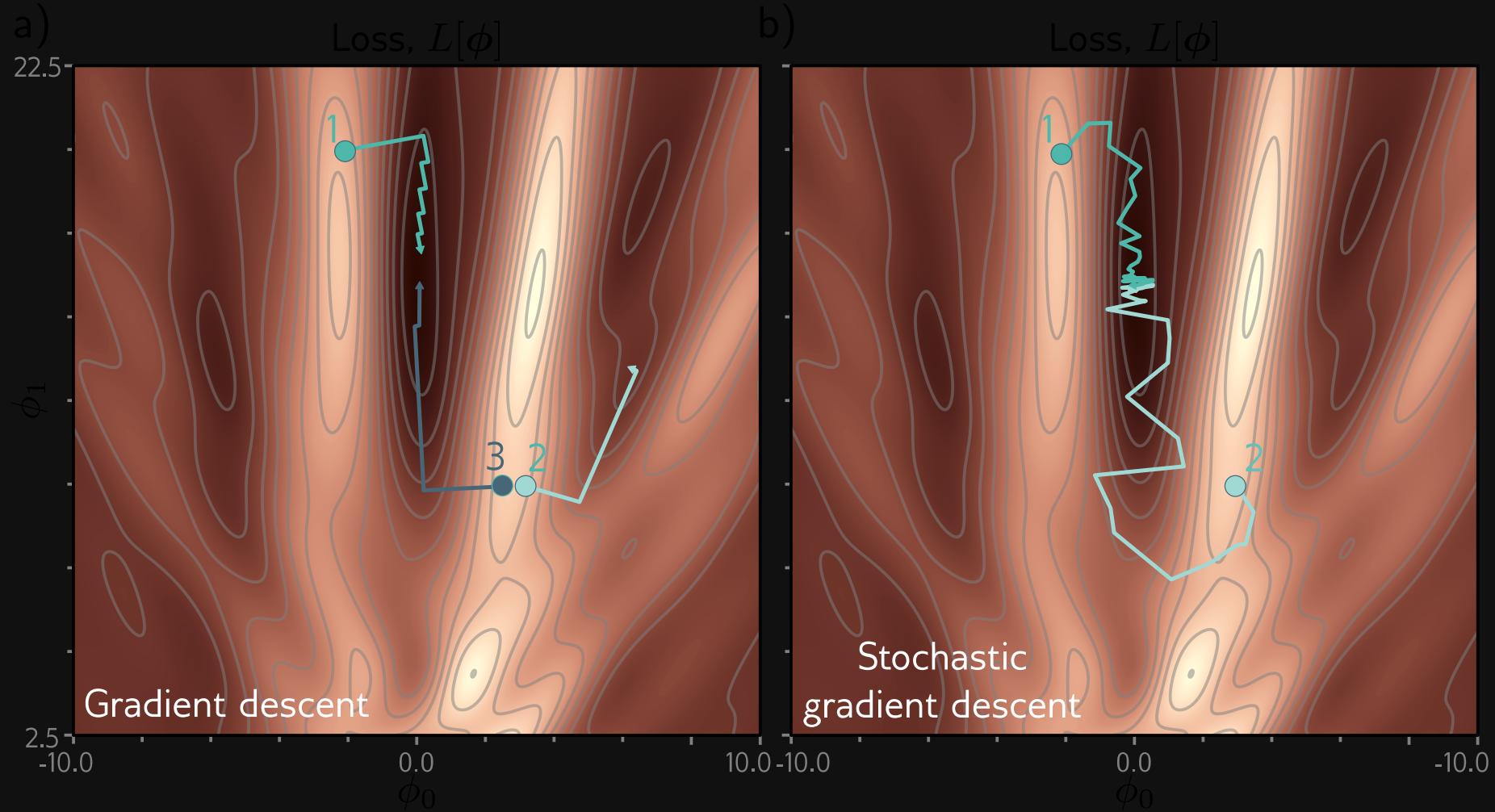
• Cauchy (1847)

Nonlinear Model

Loss functions generally non-convex, possibly multiple local minima.



Stochastic Gradient Descent

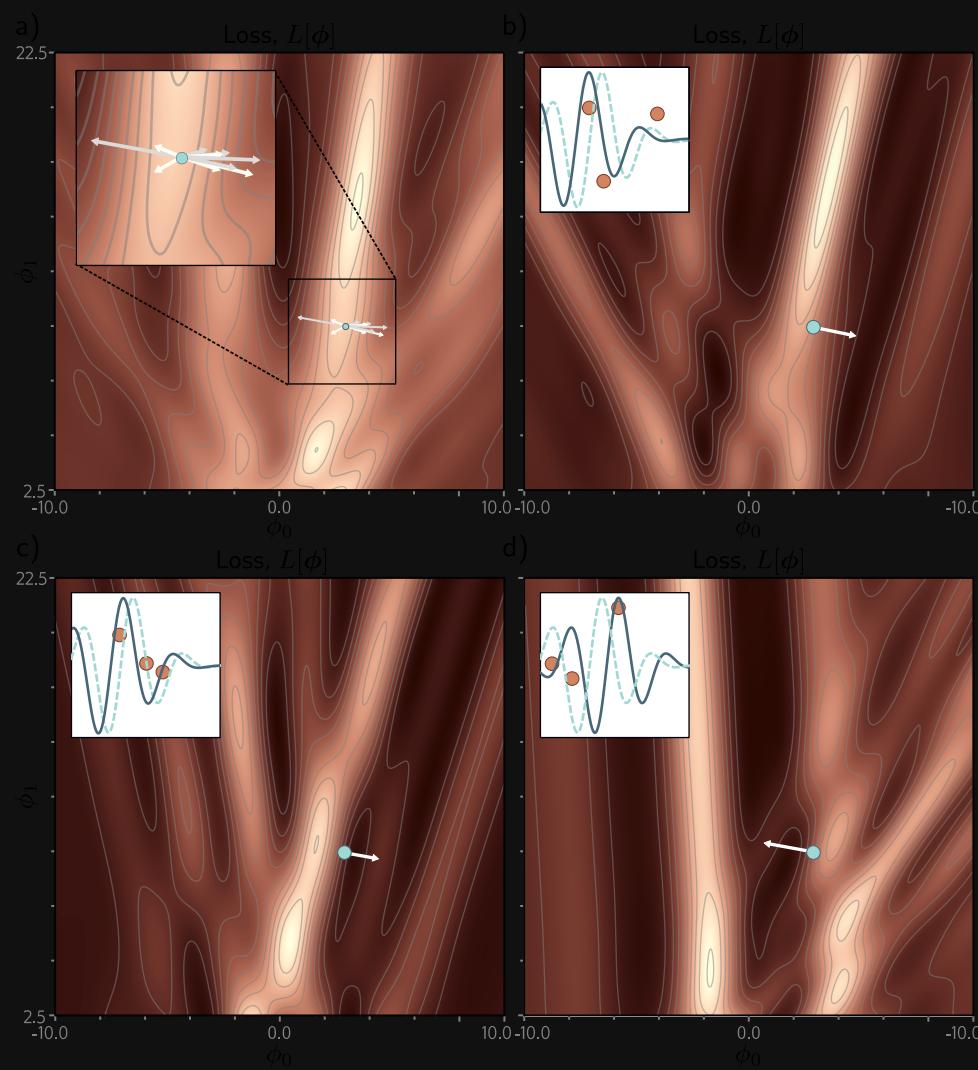


Stochastic Gradient Descent¹

Compute gradient on a mini-batch B_t of data points

$$\phi_{t+1} \leftarrow \phi_t - \alpha \sum_{i \in B_t} \nabla_\phi l_i(\phi_t; x_i, y_i)$$

- Sampled without replacement within each epoch (sweep of training data).
- Learning rate often decreasing over time.



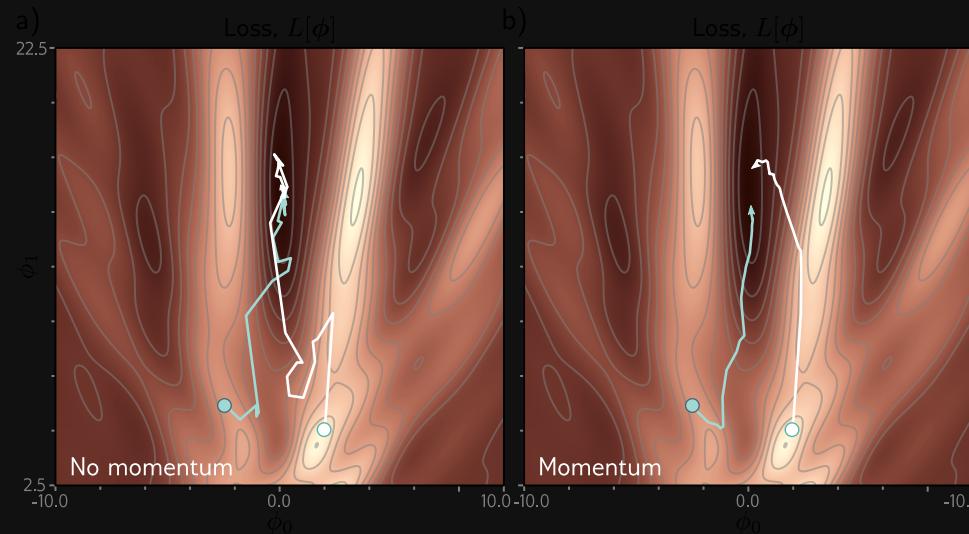
. Robbins & Monro (1951)

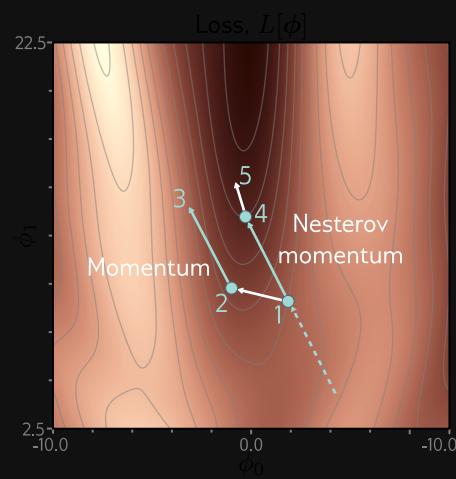
Nesterov Accelerated Momentum¹

$$m_{t+1} \leftarrow \beta m_t + (1 - \beta) \sum_{i \in B_t} \nabla_{\phi} l_i(\phi_t - \alpha \beta m_t; x_i, y_i)$$

$$\phi_{t+1} \leftarrow \phi_t - \alpha m_{t+1}$$

adds momentum (m_t) *before* evaluating gradient.





. Nesterov (1983)

Adaptive Moment Estimation (Adam)¹

Normalizes the gradient while relying on momentum to avoid convergence issues.

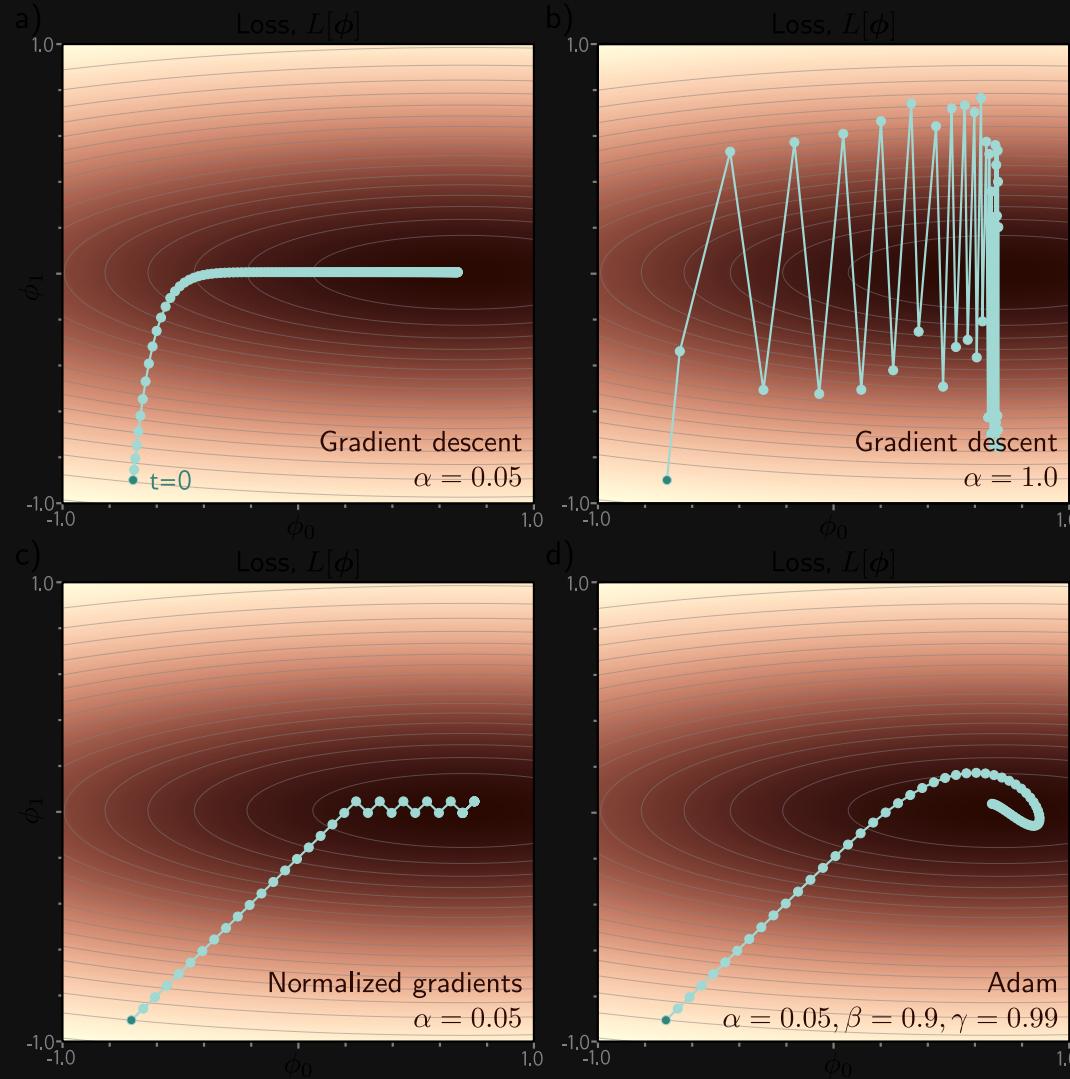
$$m_{t+1} \leftarrow \beta m_t + (1 - \beta) \sum_{i \in B_t} \nabla_{\phi} l_i(\phi_t; x_i, y_i)$$

$$v_{t+1} \leftarrow \gamma v_t + (1 - \gamma) \left(\sum_{i \in B_t} \nabla_{\phi} l_i(\phi_t; x_i, y_i) \right)^2$$

$$\phi_{t+1} \leftarrow \phi_t - \alpha \frac{\frac{m_{t+1}}{1-\beta^{t+1}}}{\sqrt{\frac{v_{t+1}}{1-\gamma^{t+1}}} + \epsilon}$$

. Kingma & Ba (2015)

Adaptive Moment Estimation (Adam)



PyTorch

PyTorch

Setup

```
1 import torch
2 torch.manual_seed(42) # for reproducibility (same results across runs)
3 # Choose where to run: MPS (Apple Silicon), CUDA (NVIDIA GPU), or CPU.
4 if torch.backends.mps.is_available():
5     DEVICE = torch.device("mps") # Apple Silicon Metal Performance Shaders
6 elif torch.cuda.is_available():
7     DEVICE = torch.device("cuda") # GPU
8 else:
9     DEVICE = torch.device("cpu") # CPU
```

Key components:

- **Tensors**: like NumPy arrays but live on CPU/GPU and integrate with autograd.
- **Autograd**: records operations so gradients are computed automatically.

Tensors

```
1 from sklearn import datasets as sk_datasets
2 X, y = sk_datasets.load_diabetes(return_X_y=True, as_frame=True)
3 X = X.iloc[-200:]
4 y = y.iloc[-200:]
5 X_train = torch.tensor(X.values, dtype=torch.float32).to(DEVICE)
6 y_train = torch.tensor(y.values, dtype=torch.float32).to("cpu")
7 print(X_train.device)
8 X_train = X_train.to("cpu")
9 print(X_train.device)
```

mps:0
cpu

Linear Regression: Closed-form Solution

```
1 X_with_intercept = torch.cat([torch.ones(X_train.shape[0], 1, dtype=X_train.dtype), X_train], dim=1)
2 solution = torch.linalg.lstsq(X_with_intercept, y_train.unsqueeze(1)) # lstsq expects RHS shape (n, 1)
3 coefficients = solution.solution.flatten() # length 11: intercept, then 10 feature coeffs
4 print("Intercept:", coefficients[0].item())
5 print("Two coefficients (e.g. bmi, bp):", coefficients[3].item(), coefficients[4].item())
```

Intercept: 152.05950927734375

Two coefficients (e.g. bmi, bp): 546.1898803710938 408.4148864746094

Linear Regression: Numerical Solution

```
1 from torch import nn
2 import torch.optim as optim
3
4 class LinearRegressionModel(nn.Module):
5     def __init__(self, input_dim, bias=True):
6         super().__init__() # calls constructor of parent class nn.Module
7         self.linear = nn.Linear(input_dim, 1, bias=bias) # one linear layer: input_dim -> 1 output
8
9     def forward(self, x):
10        return self.linear(x)
11
12 model = LinearRegressionModel(input_dim=X_train.shape[1])
13 model = model.to(DEVICE) # Model and data must be on the same device
14 X_train = X_train.to(DEVICE)
15 y_train = y_train.to(DEVICE)
16
17 criterion = nn.MSELoss()
18 optimizer = optim.Adam(model.parameters(), lr=0.01)
19
20 # Training loop (enough epochs to approach OLS solution)
21 epochs = 50000
22 for epoch in range(epochs): # one epoch = one pass over the entire training set
23     # FORWARD PASS: compute predictions and loss.
24     outputs = model(X_train) # calls LinearRegressionModel.forward(x)
```

Epoch [5000/50000], Loss: 16467.6094

Epoch [10000/50000], Loss: 8020.4341

Epoch [15000/50000], Loss: 4216.0083

Epoch [20000/50000], Loss: 3449.6143

Epoch [25000/50000], Loss: 3194.2490

Epoch [30000/50000], Loss: 3021.3784

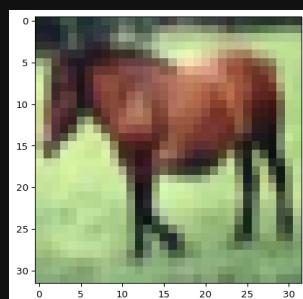
Epoch [35000/50000], Loss: 2902.3015
Epoch [40000/50000], Loss: 2827.1619
Epoch [45000/50000], Loss: 2780.8799
Epoch [50000/50000], Loss: 2756.5325
linear.weight: [[50.884987 -217.22371 471.00696 427.02527 -65.95926 -7.058841
-270.20367 44.71524 451.69968 -20.07624]]
linear.bias: [151.89984]

Datasets

```
1 from torchvision import datasets
2 from torchvision.transforms import ToTensor
3 import matplotlib.pyplot as plt
4
5 training_data = datasets.CIFAR10( # CIFAR-10 is 60k small images in 10 classes.
6     root="data",           # directory where data is stored
7     train=True,            # True = training split, False = test split
8     download=True,
9     transform=ToTensor()   # convert PIL image to tensor, scale pixels to [0, 1]
10 )
11 test_data = datasets.CIFAR10(root="data", train=False, download=True, transform=ToTensor())
12
13 image, label = training_data[7] # Indexing a Dataset returns one sample.
14 print(f"Label: {training_data.classes[label]}")
15 print(f"Image size: {image.shape}") # Image shape is (Channels, Height, Width), e.g. (3, 32, 32) for
16 plt.imshow(image.permute(1, 2, 0)) # permute to (H, W, C) for matplotlib
17 plt.show()
```

Label: horse

Image size: torch.Size([3, 32, 32])



Data Loader

```
1 from torch.utils.data import DataLoader
2 import numpy as np
3 import random
4
5 def seed_worker(worker_id): # Reproducibility: seed workers and generator so shuffle is deterministic
6     worker_seed = torch.initial_seed() % 2**32
7     np.random.seed(worker_seed)
8     random.seed(worker_seed)
9     g_seed = torch.Generator()
10    g_seed.manual_seed(torch.initial_seed() % 2**32)
11
12 train_dataloader = DataLoader(training_data, # Wrap training data in a DataLoader that yields batches
13     batch_size=64, # each iteration gives 64 samples.
14     shuffle=True, # random order each epoch.
15     num_workers=2, # parallel loading.
16     worker_init_fn=seed_worker, generator=g_seed)
17 test_dataloader = DataLoader(test_data, batch_size=64, shuffle=True, num_workers=2,
18     worker_init_fn=seed_worker, generator=g_seed)
19
20 batch_images, batch_labels = next(iter(train_dataloader)) # next batch: shapes (batch_size, C, H, W)
21 print('Batch size:', batch_images.shape)
22 plt.imshow(batch_images[0].permute(1, 2, 0))
23 plt.show()
```

Mini-batch training loop (one optimizer step per batch)

```

1 from torch.utils.data import TensorDataset, DataLoader
2
3 # Wrap tensors in a Dataset, then DataLoader
4 train_dataset = TensorDataset(X_train, y_train)
5 train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True) # 32 samples per batch, random
6
7 model_mb = LinearRegressionModel(input_dim=X_train.shape[1]).to(DEVICE)
8 criterion = nn.MSELoss()
9 optimizer_mb = optim.Adam(model_mb.parameters(), lr=0.01)
10
11 model_mb.train() # Training mode so dropout/BatchNorm (if any) behave correctly during training.
12
13 n_epochs = 500
14 for epoch in range(n_epochs):
15     epoch_loss = 0.0
16     for batch_x, batch_y in train_loader: # Batch = a subset of the training data. One forward/backwa
17         batch_x = batch_x.to(DEVICE) # size (batch_size, n_features)
18         batch_y = batch_y.to(DEVICE) # size (batch_size,)
19
20         # Forward: predictions and loss for this batch only.
21         pred = model_mb(batch_x)
22         loss = criterion(pred, batch_y.unsqueeze(1)) # target shape (batch_size, 1) to match pred
23
24         # Backward: compute gradients, then optimizer step (update parameters).

```

Epoch 100/500, mean loss: 28557.7874
 Epoch 200/500, mean loss: 25439.1797
 Epoch 300/500, mean loss: 23249.3605
 Epoch 400/500, mean loss: 22399.3983

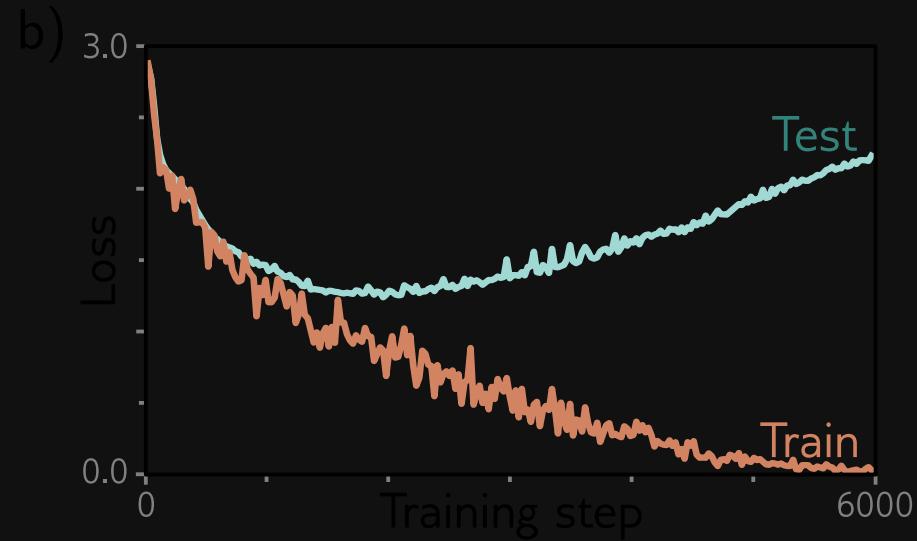
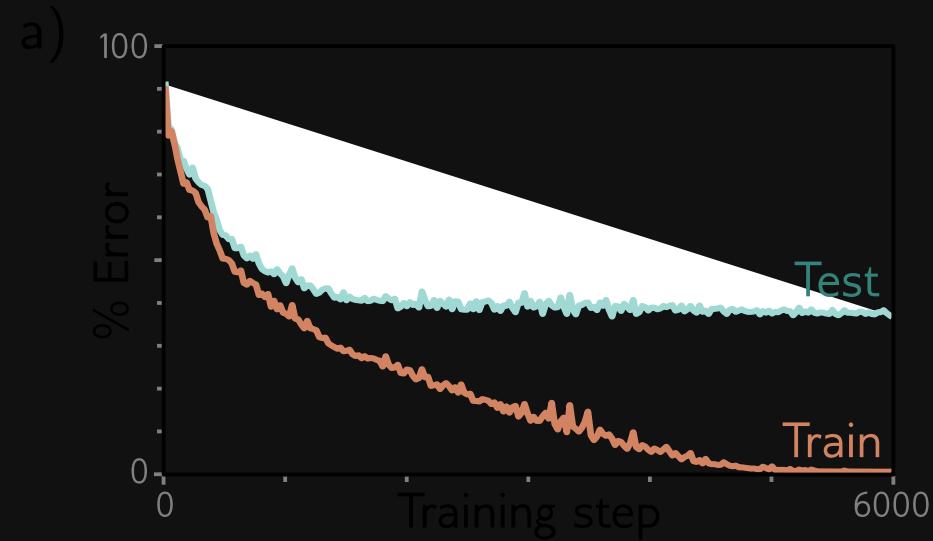
Epoch 500/500, mean loss: 21097.8613

Sample predictions (first 3): [22.177624 29.933098 28.115124]

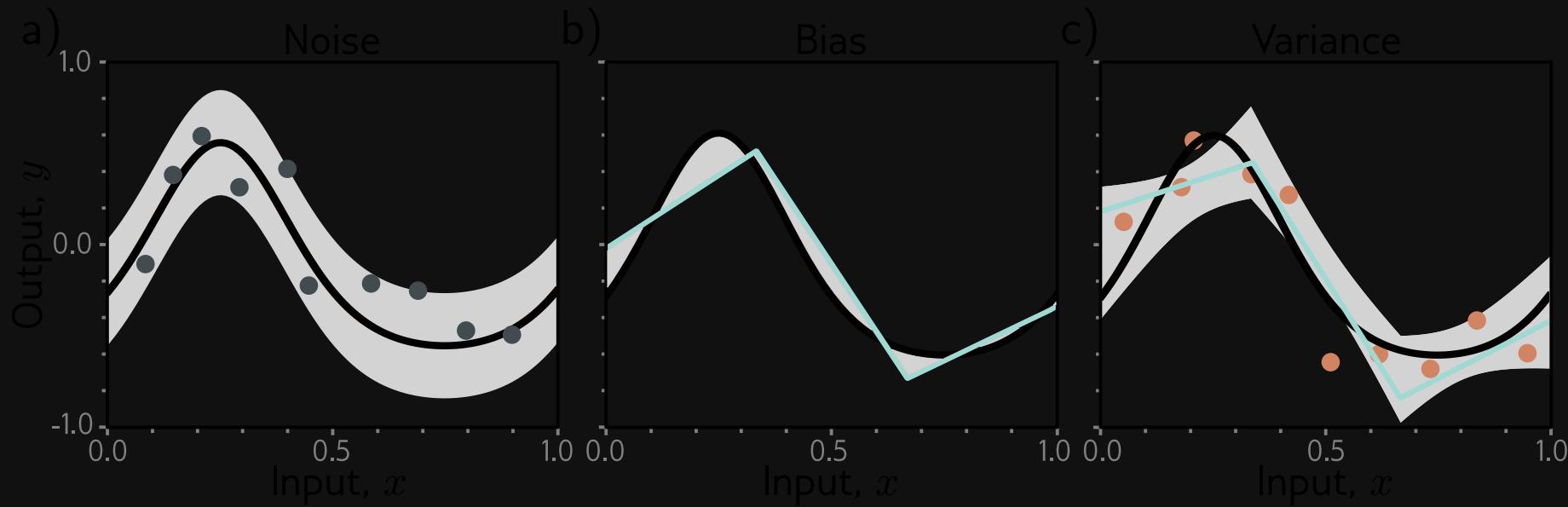
Measuring Performance

Prince (2023, chap. 8)

Training and Test Error



Error Sources



Error Decomposition

In a linear regression,

$$\begin{aligned} L[x] &= (f[x, \phi] - \mathbb{E}[y|x] + \mathbb{E}[y|x] - y[x])^2 \\ &= (f[x, \phi] - \mathbb{E}[y|x])^2 + 2(f[x, \phi] - \mathbb{E}[y|x])(\mathbb{E}[y|x] - y[x]) + (\mathbb{E}[y|x] - y[x])^2 \end{aligned}$$

$y[x]$ and hence L are stochastic, so taking expectations w.r.t. test data y

$$\begin{aligned} \mathbb{E}_y[L[x]] &= (f[x, \phi] - \mathbb{E}[y|x])^2 + 2(f[x, \phi] - \mathbb{E}[y|x])(\mathbb{E}[y|x] - \mathbb{E}[y|x]) + (\mathbb{E}[y|x] - \mathbb{E}[y|x])^2 \\ &= (f[x, \phi] - \mathbb{E}[y|x])^2 + 0 + \sigma^2 \end{aligned}$$

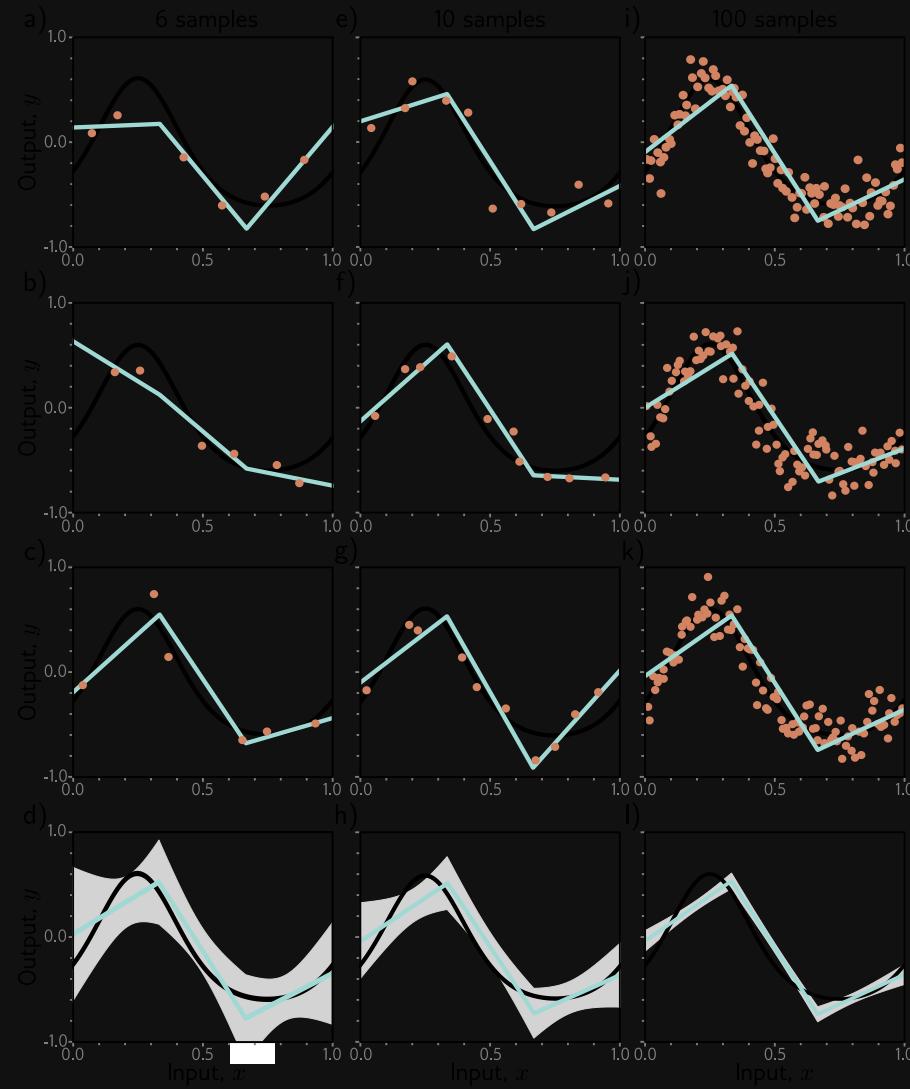
Taking expectations with respect to training data D

$$\begin{aligned} \mathbb{E}_D [\mathbb{E}_y [L[x]]] &= \mathbb{E}_D [(f[x, \phi[D]] - \mathbb{E}_D [f[x, \phi[D]]])^2] + (\mathbb{E}_D [f[x, \phi[D]]] - \mathbb{E}_D [\mathbb{E}_y [f[x, \phi[D]]]])^2 \\ &= \text{Variance} + \text{Bias}^2 + \text{Noise} \end{aligned}$$

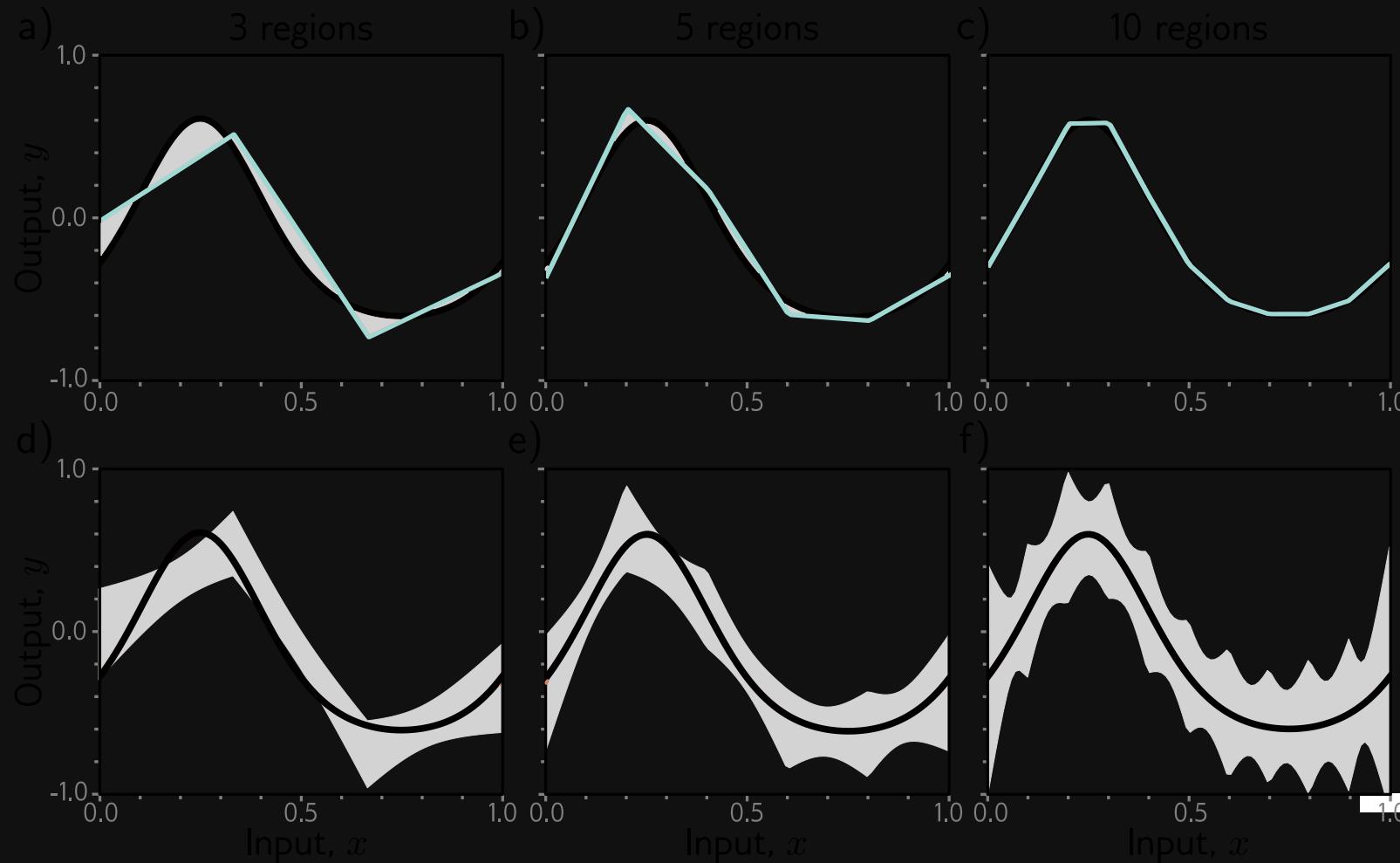
where

- variance due to sample noise in training data
- bias due to model misspecification
- noise due to inherent uncertainty in $y|x$

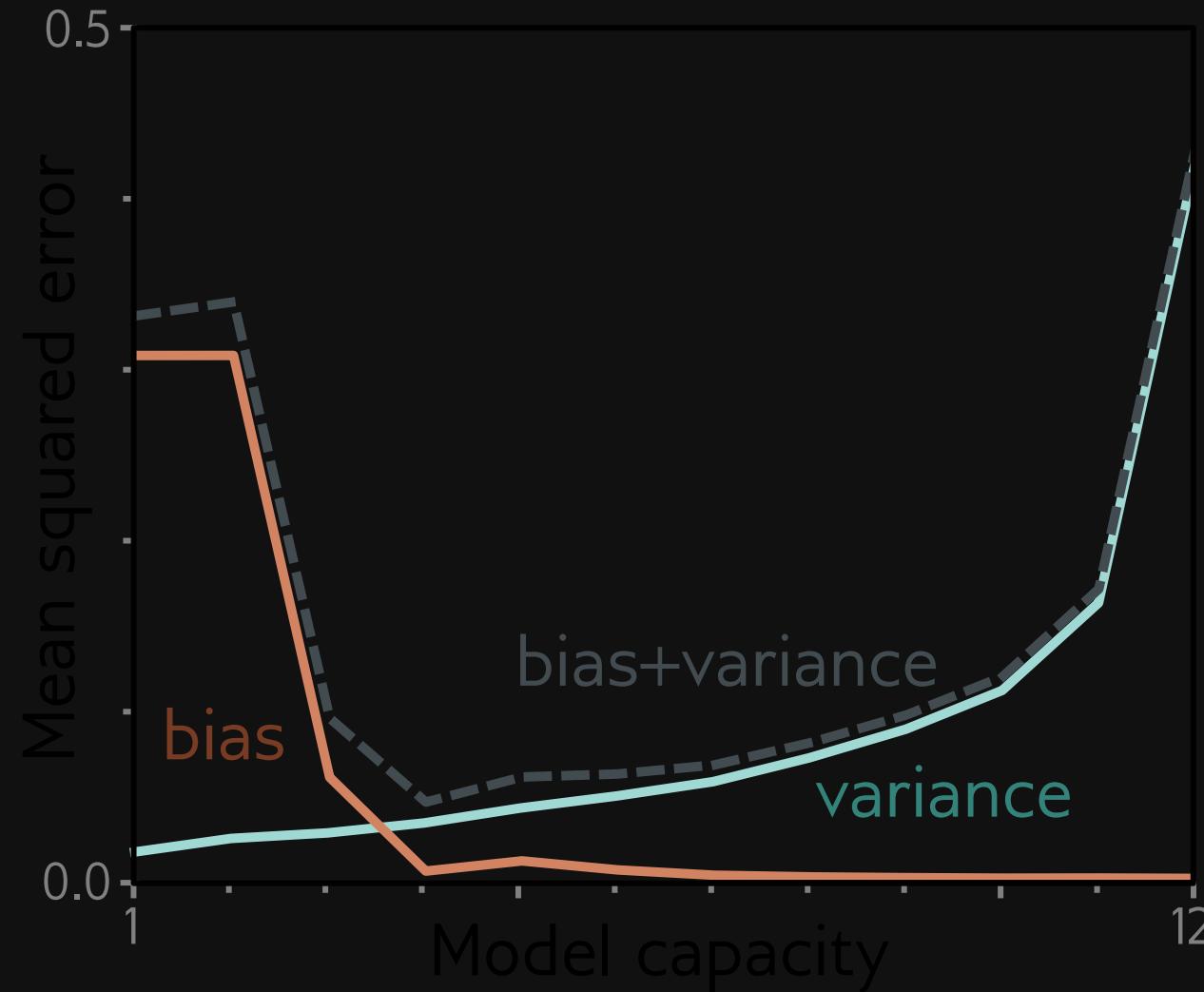
More Training Data: Less Variance



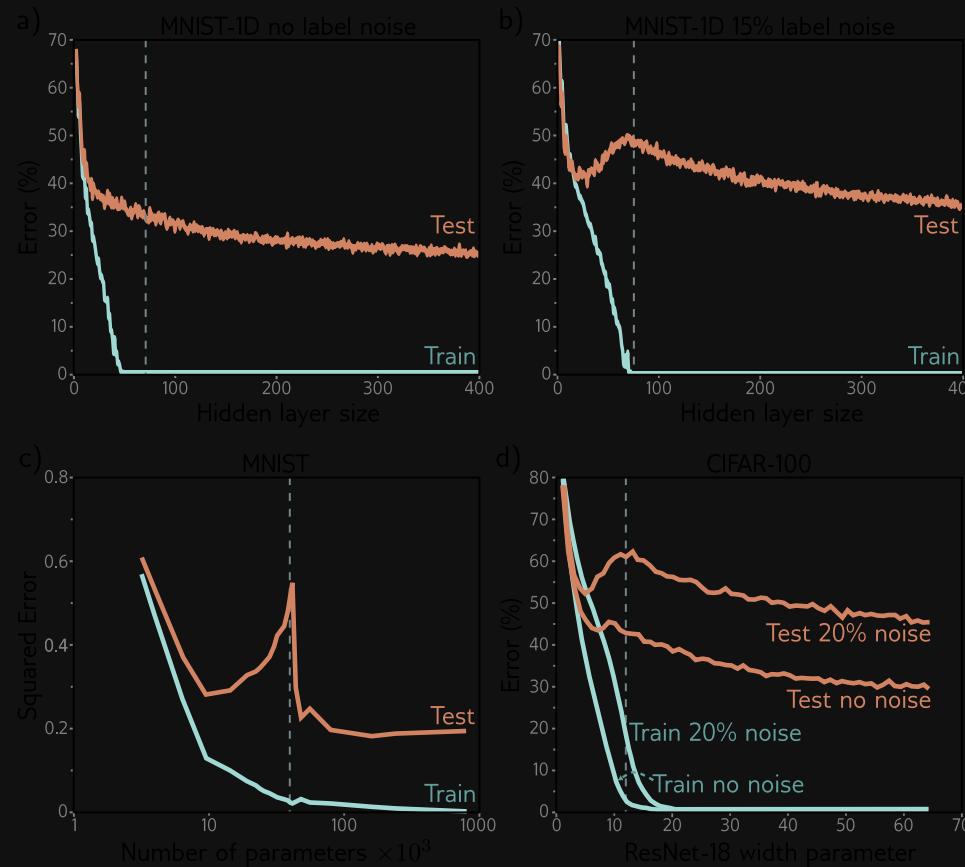
Increasing Model Capacity: Less Bias but More Variance



Bias-Variance Tradeoff



Double Descent (Overparametrization)



Overparametrization permits (and initialization and/or fitting may encourage) smoother interpolation between training data (implicit

Cross-Validation

- Learn model parameters on training set
- Choose hyperparameters using validation set
- Evaluate final model on test set

Classification: Discrete Choice

Classification

- Linear regression: continuous outcome (e.g. house price, earnings, etc.)
- Classification: discrete outcome (e.g. product choice, default vs no default, employment status)

Discrete Choice

- Outcome: $y_i \in \{1, \dots, J\}$ (choice among J alternatives), covariates \mathbf{X}_i
- Goal: model $P(y_i = j|\mathbf{X}_i)$

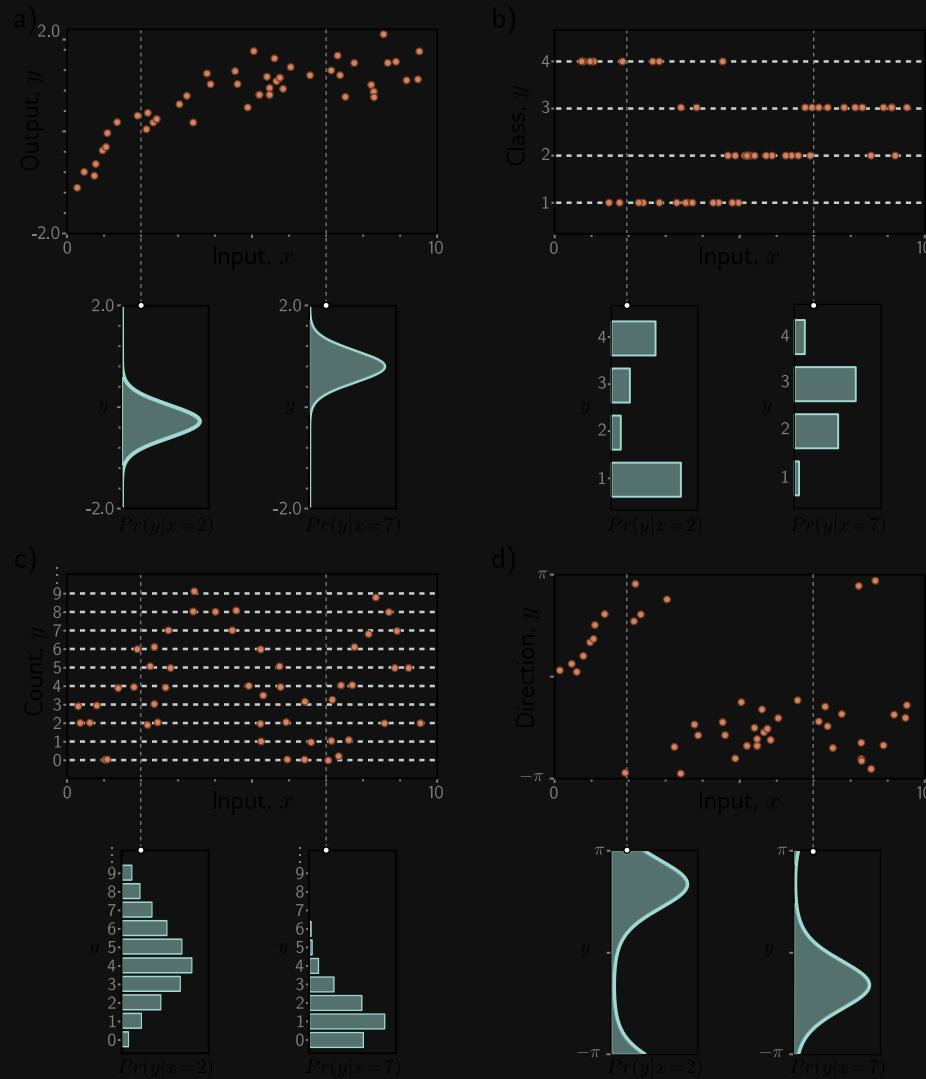
Random Utility Model and Multinomial Logit

- **Random utility**¹: agent chooses alternative with highest utility
- $U_{ij} = \mathbf{X}_i \beta_j + \varepsilon_{ij}$; choose j iff $U_{ij} > U_{ia}$ for all $a \neq j$
- If ε_{ij} i.i.d. **Gumbel** (extreme value type I), choice probabilities have a closed form (**multinomial logit**, or softmax)

$$P(y_i = j | \mathbf{X}_i) = \frac{\exp(\mathbf{X}_i \beta_j)}{\sum_{a=1}^J \exp(\mathbf{X}_i \beta_a)}$$

. McFadden (1974); Nobel Prize (2000)

Regression vs. Classification

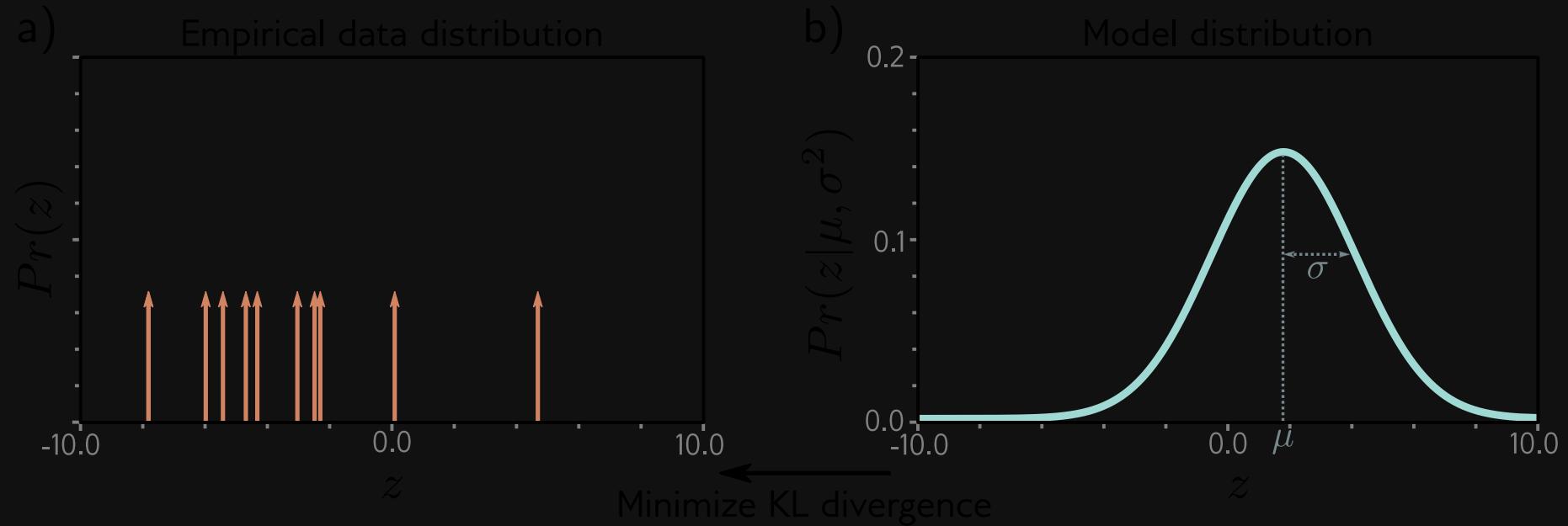


Loss Functions

Prince (2023, chap. 5)

What is a Good Model?

Model (distribution) Q that is *close* to the true distribution P (of which we observe an empirical sample).



Cross-Entropy Loss (= Max LLH)

Minimizing KL div. from model $Q_\theta(y|x)$ to empirical $P(y|x)$

$$\begin{aligned}
 \hat{\theta} &= \arg \min_{\theta} D_{KL}[P||Q] \\
 &= \arg \min_{\theta} \int_{-\infty}^{\infty} p(y|x) \log \frac{p(y|x)}{q_{\theta}(y|x)} dy \\
 &= \arg \min_{\theta} - \int_{-\infty}^{\infty} p(y|x) \log q_{\theta}(y|x) dy \\
 &= \arg \min_{\theta} - \int_{-\infty}^{\infty} \left(\frac{1}{I} \sum_{i=1}^I \delta[y - y_i] \right) \log q_{\theta}(y|x) dy \\
 &= \arg \min_{\theta} - \sum_{i=1}^I \log q_{\theta}(y_i|x_i)
 \end{aligned}$$

is equivalent to minimizing the negative log-likelihood

$$\hat{\phi} = \arg \min_{\phi} - \sum_{i=1}^I \log q(y_i | \theta_i = f[x_i, \phi])$$

Distribution parametrized by ML model.

Maximizing Log-Likelihood

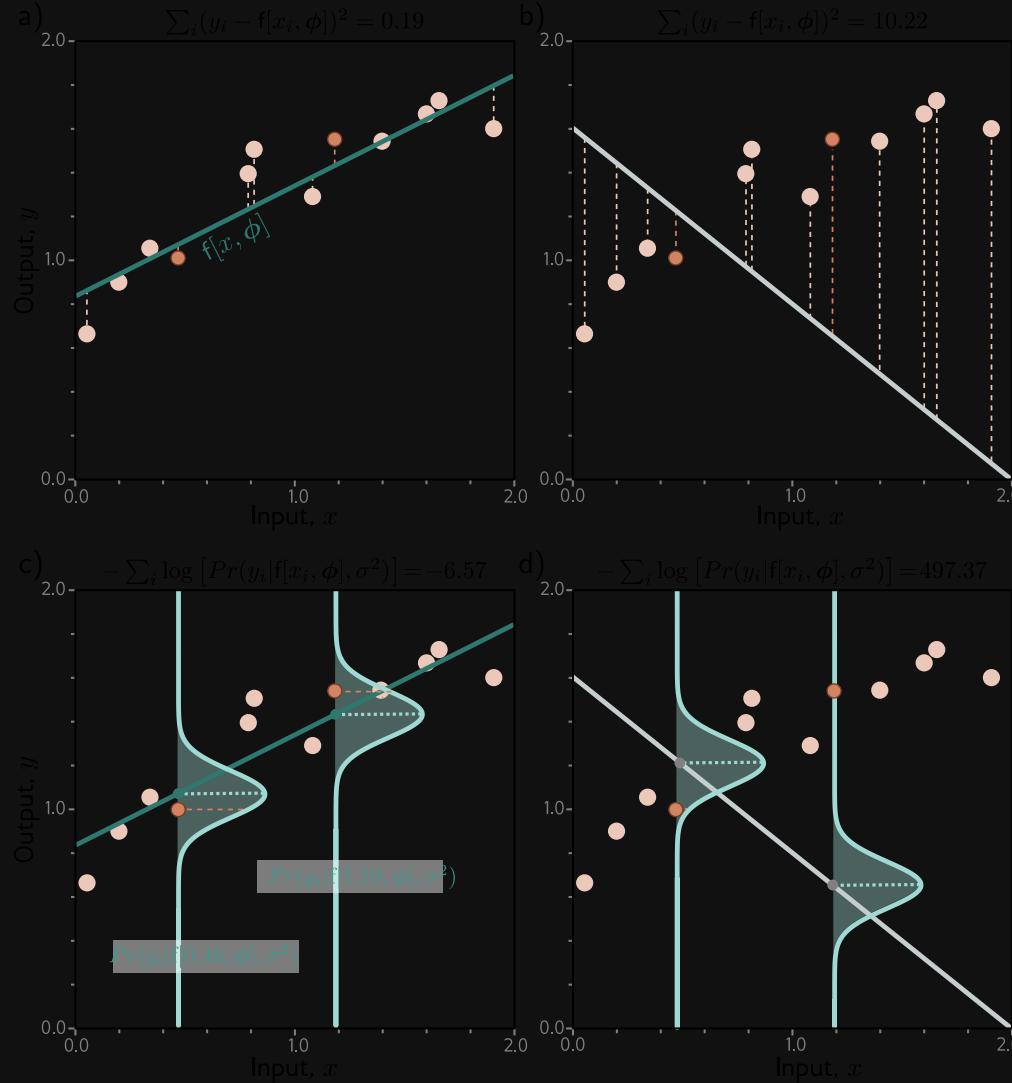
$$\begin{aligned}\hat{\phi} &= \arg \max_{\phi} \prod_{i=1}^I q(y_i | \theta_i = f[x_i, \phi]) \\ &= \arg \max_{\phi} \log \prod_{i=1}^I q(y_i | \theta_i = f[x_i, \phi]) \\ &= \arg \min_{\phi} - \sum_{i=1}^I \log q(y_i | \theta_i = f[x_i, \phi])\end{aligned}$$

Lin. Reg.: Least Squares Loss

Assuming $y_i | x_i \sim N(f[x_i, \phi], \sigma^2)$

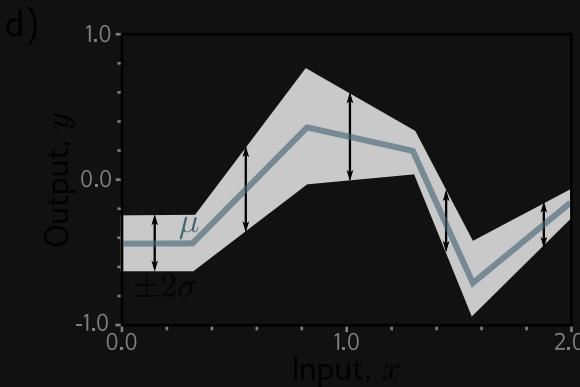
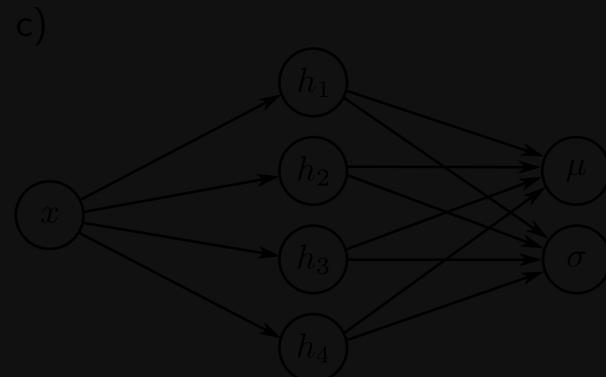
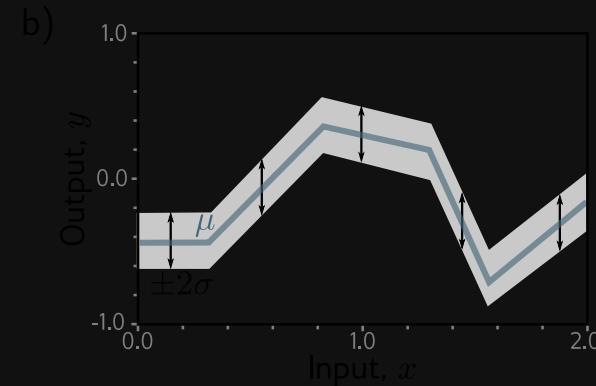
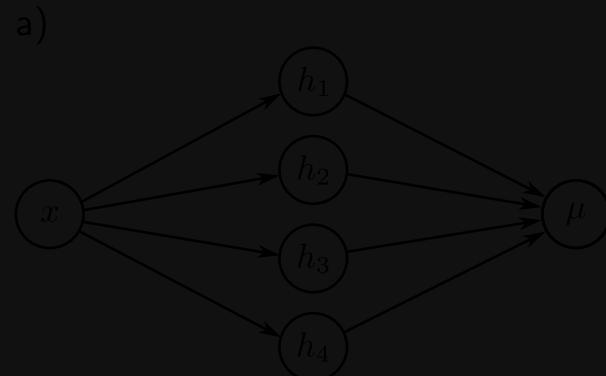
$$\begin{aligned}
 \hat{\phi} &= \arg \min_{\phi} - \sum_{i=1}^I \log q(y_i | \theta_i = f[x_i, \phi]) \\
 &= \arg \min_{\phi} - \sum_{i=1}^I \log \left(\frac{1}{\sqrt{2\pi\sigma^2}} \exp \left(-\frac{(y_i - f[x_i, \phi])^2}{2\sigma^2} \right) \right) \\
 &= \arg \min_{\phi} \sum_{i=1}^I - \log \left(\frac{1}{\sqrt{2\pi\sigma^2}} \right) + \frac{(y_i - f[x_i, \phi])^2}{2\sigma^2} \\
 &= \arg \min_{\phi} \sum_{i=1}^I (y_i - f[x_i, \phi])^2
 \end{aligned}$$

Lin. Reg.: Least Squares Loss



Heteroscedastic Regression

$$\arg \min_{\phi} - \sum_{i=1}^I \log \left(\frac{1}{\sqrt{2\pi f_2[x_i, \phi]^2}} \exp \left(-\frac{(y_i - f_1[x_i, \phi])^2}{2f_2[x_i, \phi]^2} \right) \right)$$



Binary Classification: Binary Cross-Entropy Loss

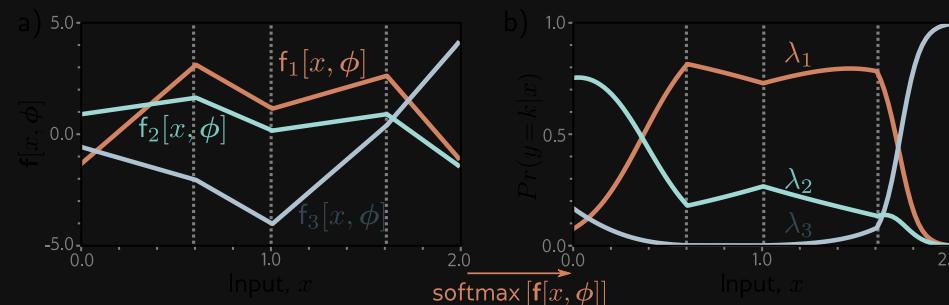
Assuming $y_i | x_i \sim Ber(\lambda_i = logit(f[x_i, \phi]))$

$$\begin{aligned}\hat{\phi} &= \arg \min_{\phi} - \sum_{i=1}^I \log((1 - logit(f[x_i, \phi])))^{1-y_i} logit(f[x_i, \phi])^{y_i} \\ &= \arg \min_{\phi} \sum_{i=1}^I -(1 - y_i) \log(1 - logit(f[x_i, \phi])) - y_i \log(logit(f[x_i,\phi]))\end{aligned}$$

Multiclass Classification: Multiclass Cross-Entropy Loss

Assuming $q(y_i = k|x_i) = \text{softmax}_k(f[x_i, \phi])$

$$\begin{aligned}\hat{\phi} &= \arg \min_{\phi} - \sum_{i=1}^I \log(\text{softmax}_{y_i}(f[x_i, \phi])) \\ &= \arg \min_{\phi} - \sum_{i=1}^I \left(f_{y_i}[x_i, \phi] - \log \left(\sum_{k'=1}^K \exp(f_{k'}[x_i, \phi]) \right) \right)\end{aligned}$$



Multivariate Output

Assuming independent dimensions $q(y|f[x, \phi]) = \prod_d q(y_d|f_d[x, \phi])$

$$\begin{aligned}\hat{\phi} &= \arg \min_{\phi} - \sum_{i=1}^I \log \left(\prod_d q(y_{i,d}|f_d[x_i, \phi]) \right) \\ &= \arg \min_{\phi} - \sum_{i=1}^I \sum_d \log(q(y_{i,d}|f_d[x_i, \phi]))\end{aligned}$$

References

Prince, Simon J. D. 2023. *Understanding Deep Learning*. Cambridge, Massachusetts: The MIT Press.