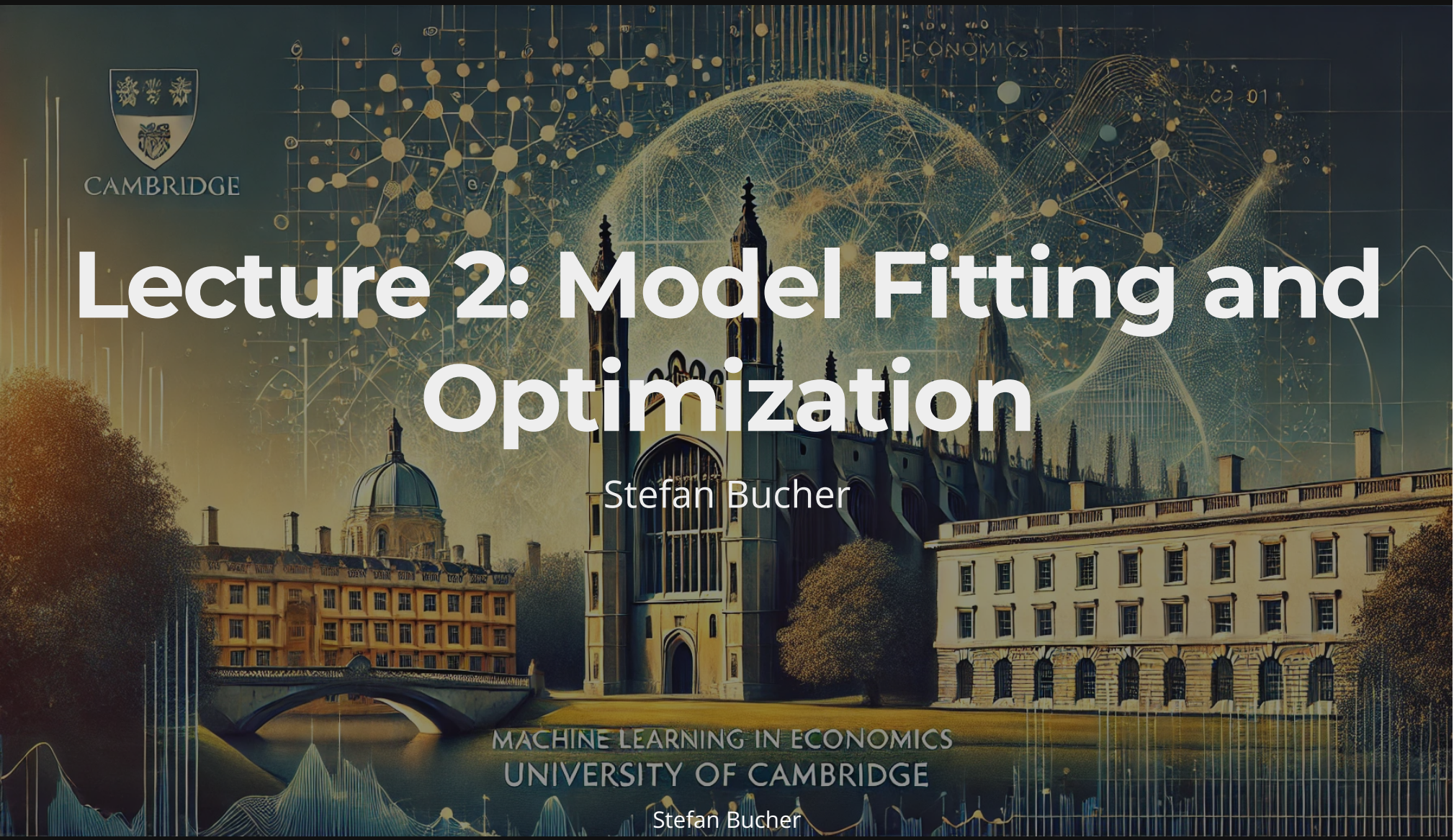


Lecture 2: Model Fitting and Optimization

Stefan Bucher

Open in Colab

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 that minimize the loss function

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

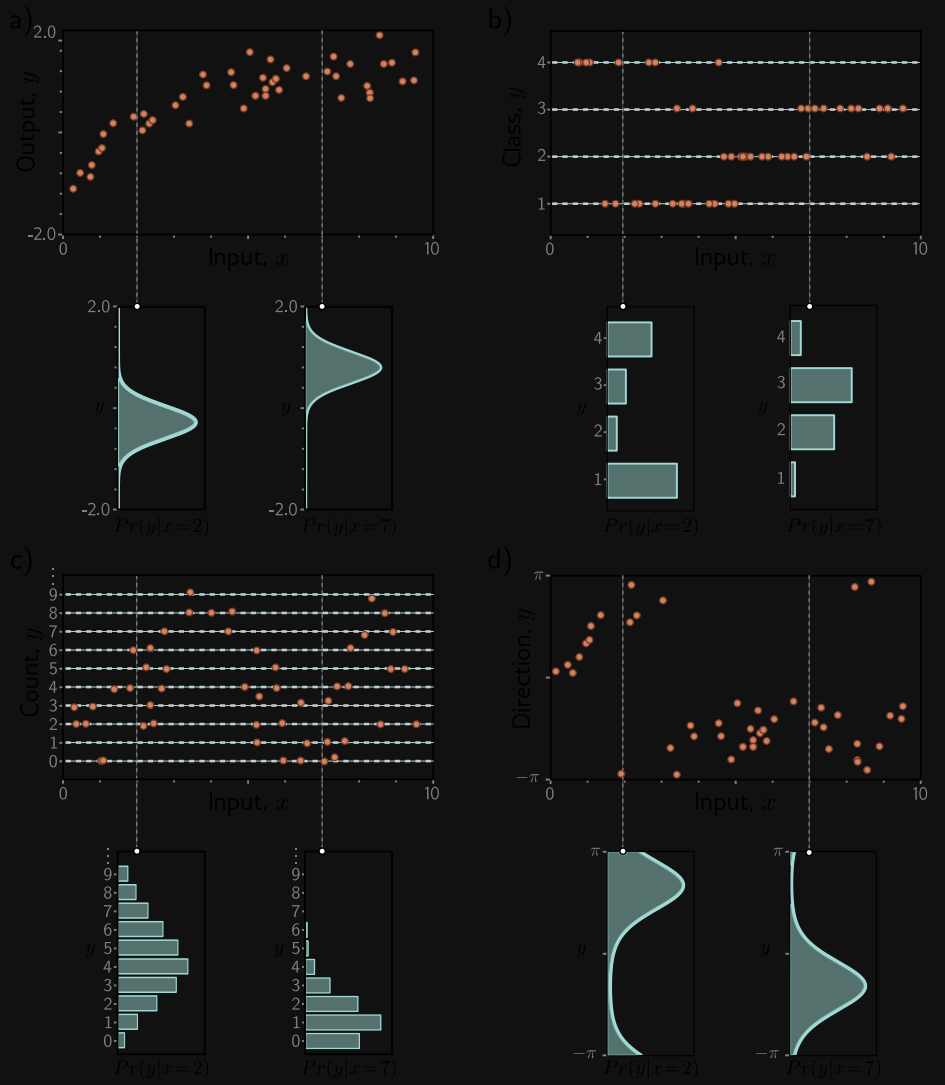
- Outcome: (choice among alternatives), covariates
- Goal: model

Random Utility Model and Multinomial Logit

- **Random utility**[1]: agent chooses alternative with highest utility
- ; choose iff for all
- If i.i.d. **Gumbel** (extreme value type I), choice probabilities have a closed form (**multinomial logit**, or softmax)

. McFadden (1974); Nobel Prize (2000)

Regression vs. Classification

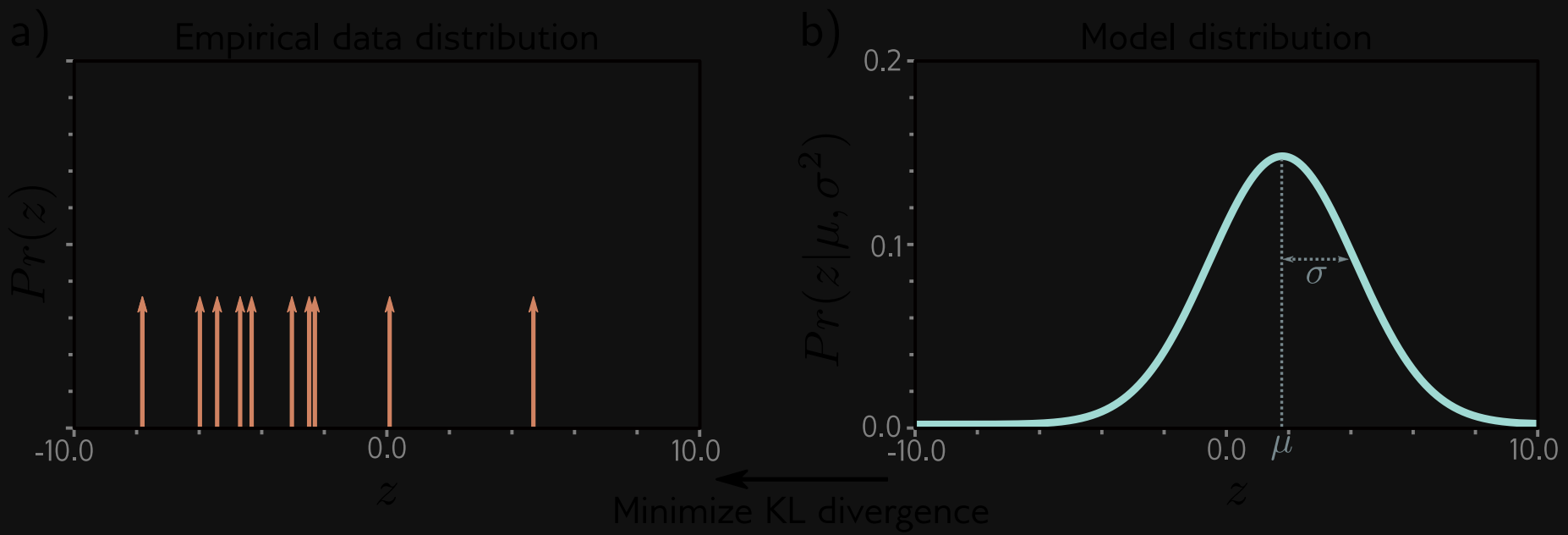


Loss Functions

Prince (2023, chap. 5)

What is a Good Model?

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



Cross-Entropy Loss (= Max LLH)

Minimizing KL div. from model to empirical is equivalent to minimizing the negative log-likelihood

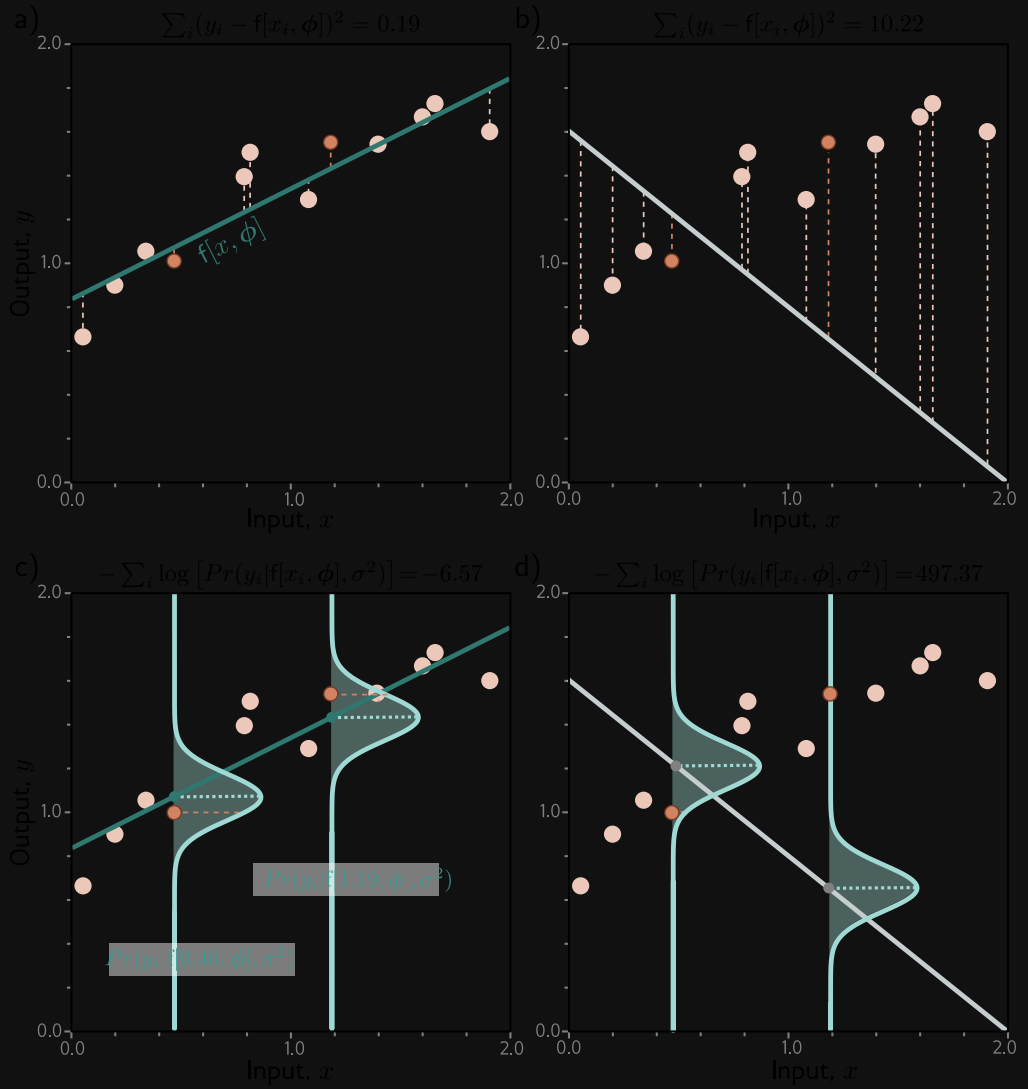
Distribution parametrized by ML model.

Maximizing Log-Likelihood

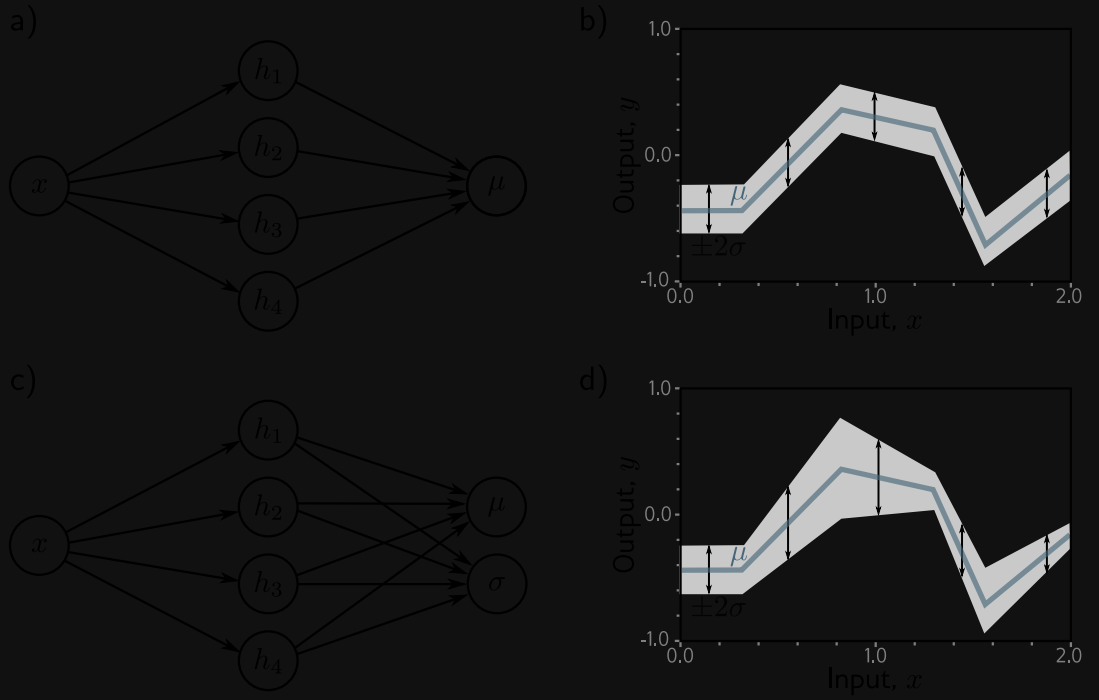
Lin. Reg.: Least Squares Loss

Assuming

Lin. Reg.: Least Squares Loss



Heteroscedastic Regression

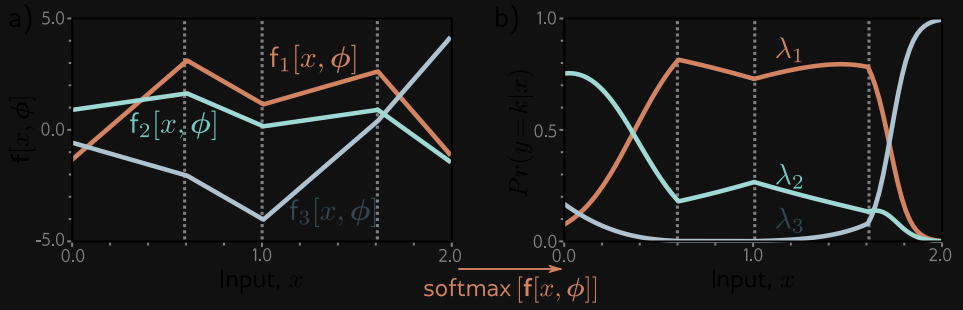


Classification: Binary Cross-Entropy Loss

Assuming

Multiclass Cross-Entropy Loss

Assuming



Multivariate Output

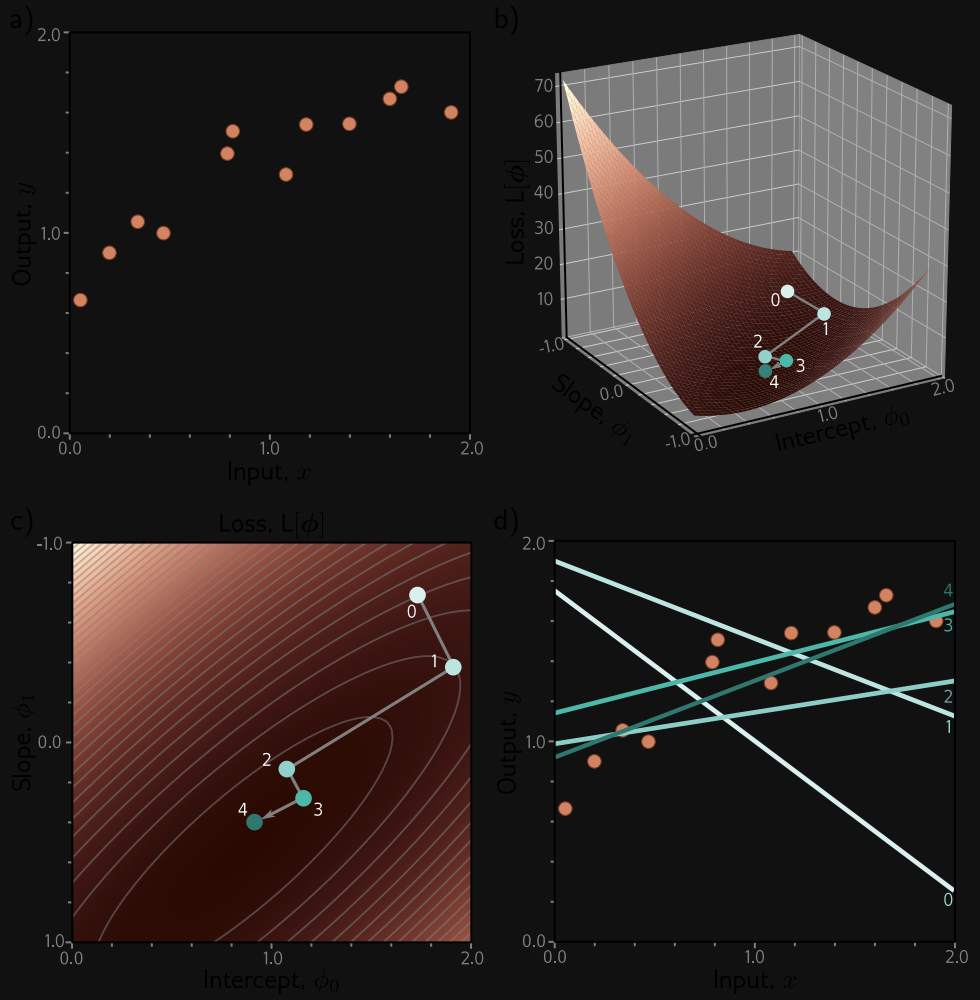
Assuming independent dimensions

Model Fitting & Optimization

Prince (2023, chap. 6)

Gradient Descent[1]

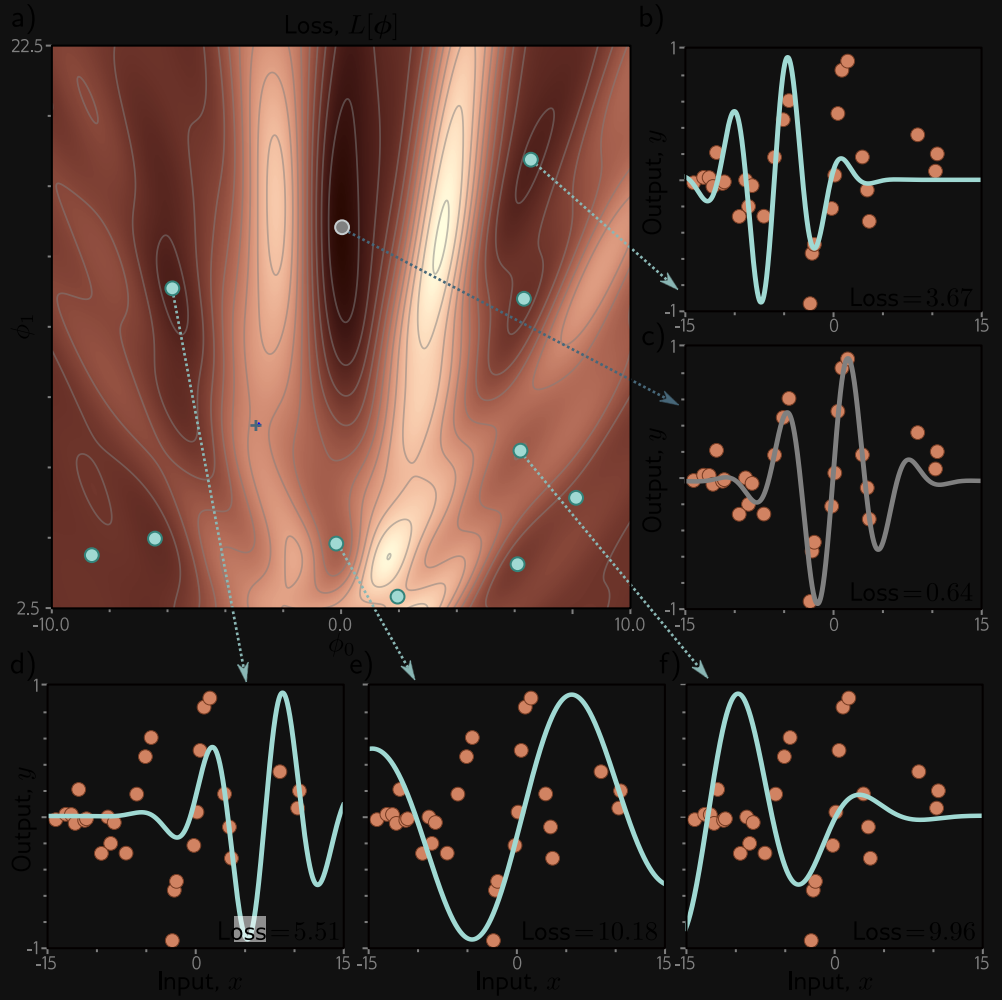
where fixed learning rate or line search.



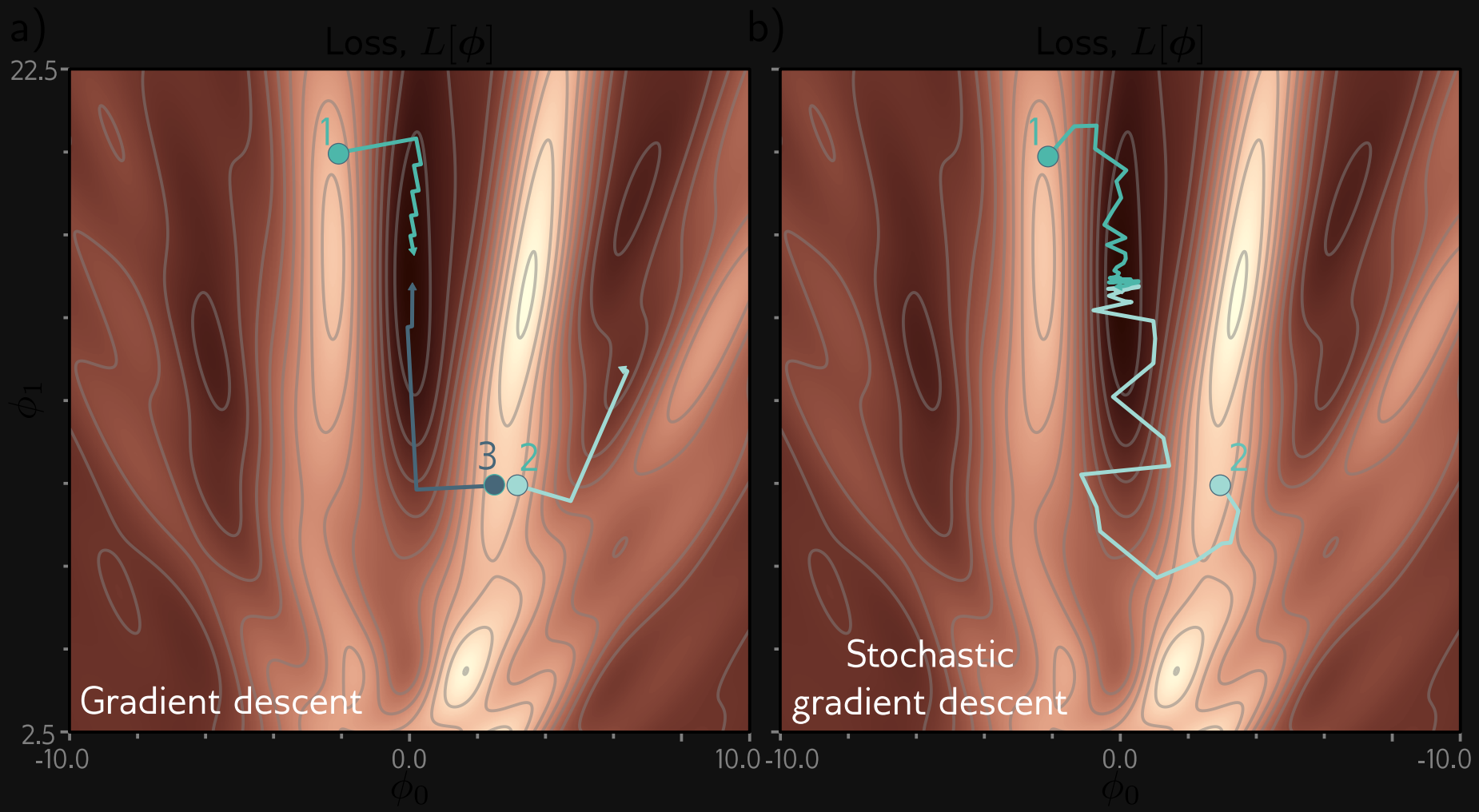
Cauchy (1847)

Nonlinear Model

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



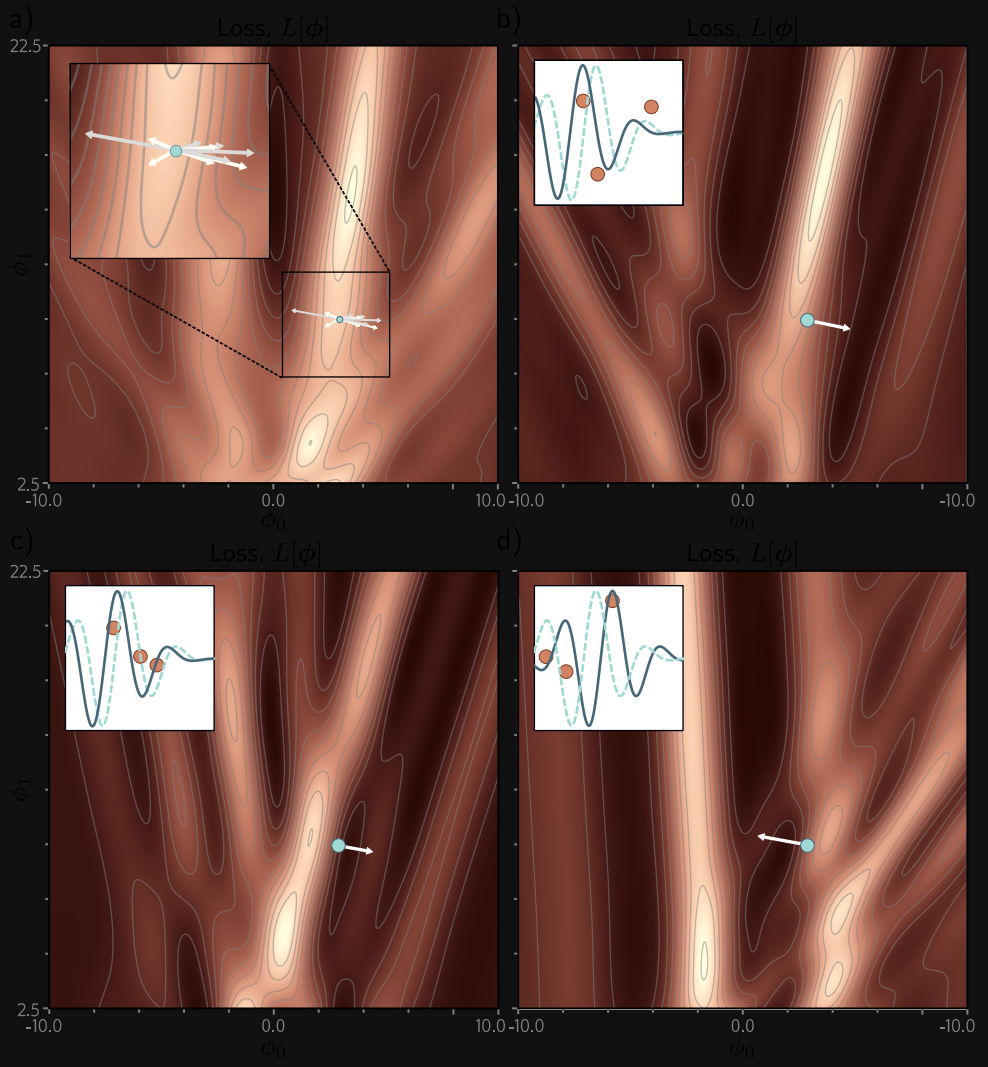
Stochastic Gradient Descent



Stochastic Gradient Descent [1]

Compute gradient on a mini-batch of data points

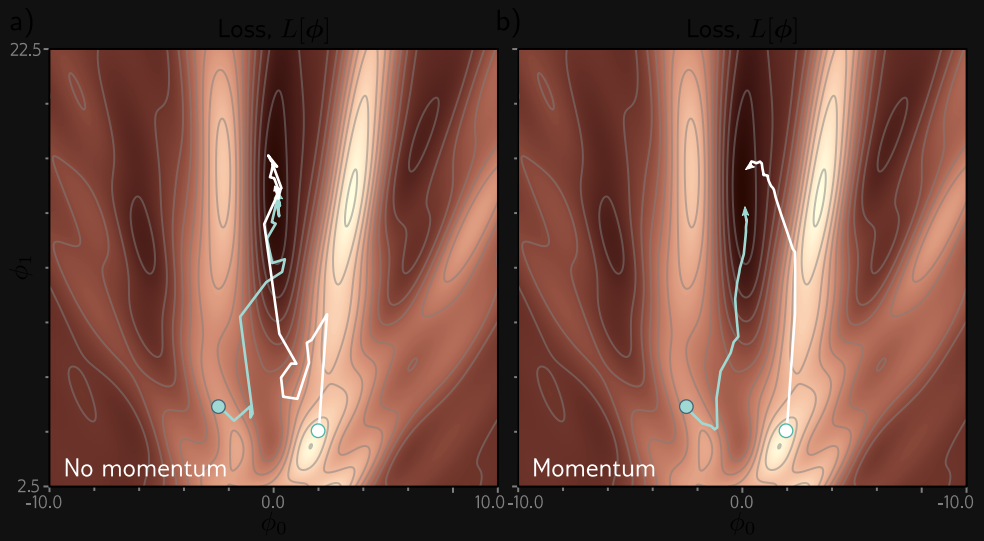
- Sampled without replacement within each epoch (sweep of training data).
- Learning rate often decreasing over time.

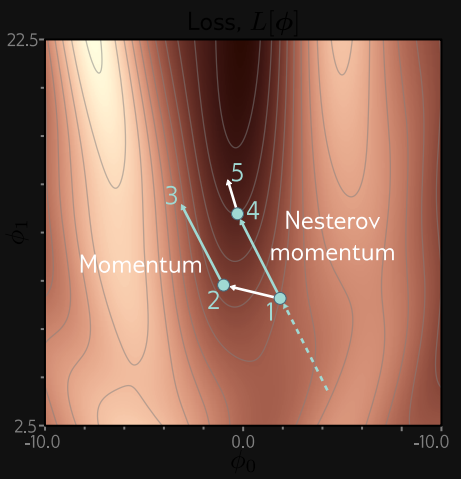


. Robbins & Monro (1951)

Nesterov Accelerated Momentum [1]

adds momentum () *before* evaluating gradient.





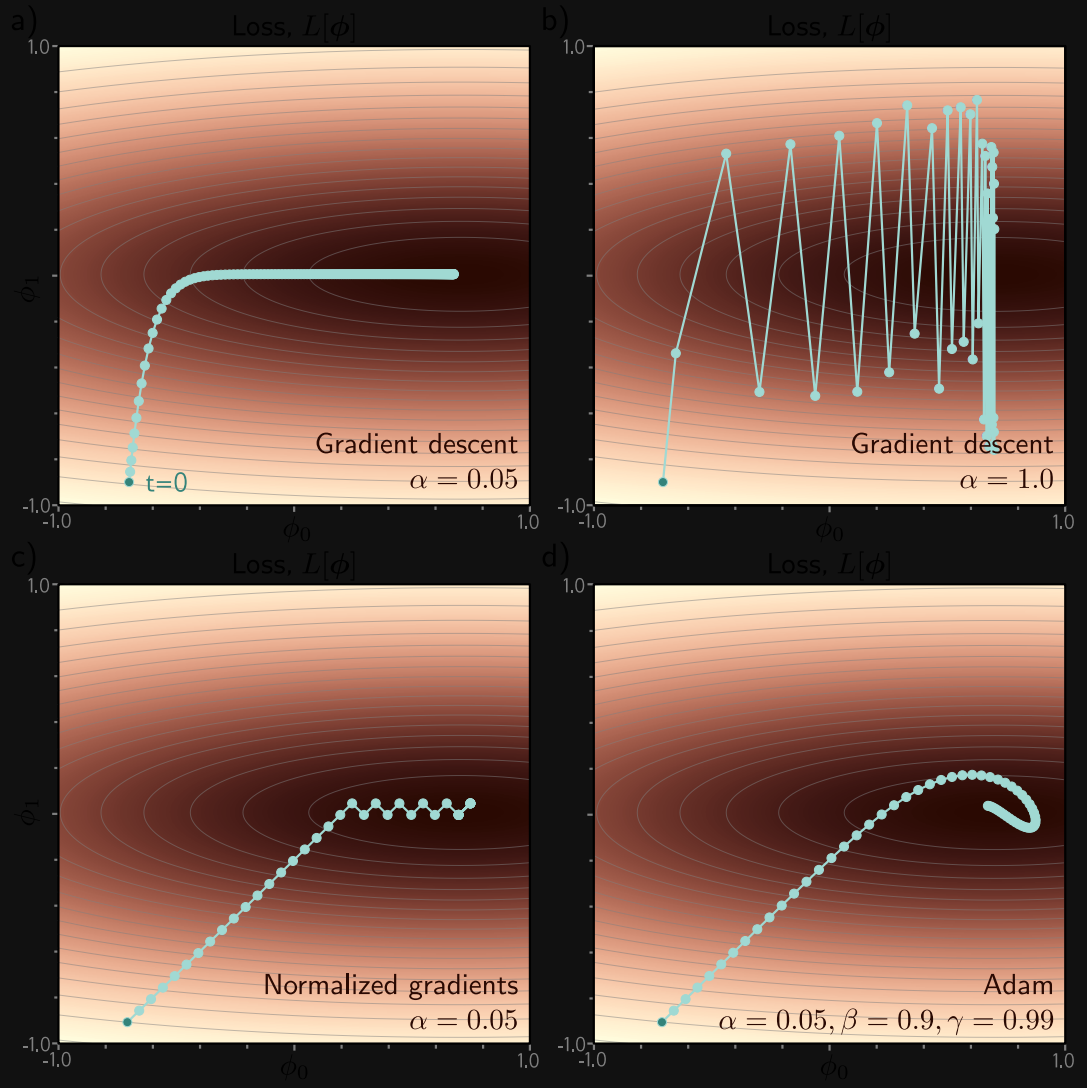
. Nesterov (1983)

Adaptive Moment Estimation (Adam) [1]

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

. Kingma & Ba (2015)

Adaptive Moment Estimation (Adam)



PyTorch

PyTorch

Setup

```python
import torch
torch.manual_seed(42)  # for reproducibility (same results across runs)
# Choose where to run: MPS (Apple Silicon), CUDA (NVIDIA GPU), or CPU.
if torch.backends.mps.is_available():
  DEVICE = torch.device("mps")  # Apple Silicon Metal Performance Shaders
elif torch.cuda.is_available():
  DEVICE = torch.device("cuda")  # GPU
else:
  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

```
X_with_intercept = torch.cat([torch.ones(X_train.shape[0], 1, dtype=X_train.dtype), X_train], dim=1)
solution = torch.linalg.lstsq(X_with_intercept, y_train.unsqueeze(1)) # lstsq expects RHS shape (n, 1
coefficients = solution.solution.flatten()   # length 11: intercept, then 10 feature coefs
print("Intercept:", coefficients[0].item())
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

```python
from torch import nn
import torch.optim as optim

class LinearRegressionModel(nn.Module):
    def __init__(self, input_dim, bias=True):
        super().__init__()  # calls constructor of parent class nn.Module
        self.linear = nn.Linear(input_dim, 1, bias=bias) # one linear layer: input_dim -> 1 output

    def forward(self, x):
        return self.linear(x)

model = LinearRegressionModel(input_dim=X_train.shape[1])
model = model.to(DEVICE) # Model and data must be on the same device
X_train = X_train.to(DEVICE)
y_train = y_train.to(DEVICE)

criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)

# Training loop
epochs = 50000
for epoch in range(epochs): # one epoch = one pass over the entire training set
    # FORWARD PASS: compute predictions and loss.
    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]
```

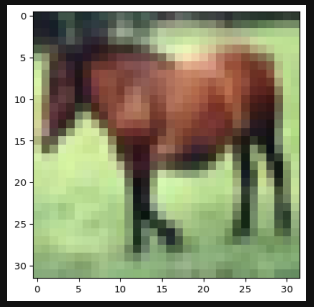
Datasets

```python
from torchvision import datasets
from torchvision.transforms import ToTensor
import matplotlib.pyplot as plt

training_data = datasets.CIFAR10( #  CIFAR-10 is 60k small images in 10 classes.
    root="data",           # directory where data is stored
    train=True,            # True = training split, False = test split
    download=True,
    transform=ToTensor()   # convert PIL image to tensor, scale pixels to [0, 1]
)
test_data = datasets.CIFAR10(root="data", train=False, download=True, transform=ToTensor())

image, label = training_data[7] # Indexing a Dataset returns one sample.
print(f"Label: {training_data.classes[label]}")
print(f"Image size: {image.shape}")   # Image shape is (Channels, Height, Width), e.g. (3, 32, 32) fo
plt.imshow(image.permute(1, 2, 0))    # permute to (H, W, C) for matplotlib
plt.show()
```

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



Data Loader

```python
from torch.utils.data import DataLoader
import numpy as np
import random

def seed_worker(worker_id): # Reproducibility: seed workers and generator so shuffle is deterministic
  worker_seed = torch.initial_seed() % 2**32
  np.random.seed(worker_seed)
  random.seed(worker_seed)
g_seed = torch.Generator()
g_seed.manual_seed(torch.initial_seed() % 2**32)

train_dataloader = DataLoader(training_data, # Wrap training data in a DataLoader that yields batches
  batch_size=64, # each iteration gives 64 samples.
  shuffle=True, # random order each epoch.
  num_workers=2, # parallel loading.
  worker_init_fn=seed_worker, generator=g_seed)
test_dataloader = DataLoader(test_data, batch_size=64, shuffle=True, num_workers=2,
  worker_init_fn=seed_worker, generator=g_seed)

batch_images, batch_labels = next(iter(train_dataloader)) # next batch: shapes (batch_size, C, H, W)
print('Batch size:', batch_images.shape)
plt.imshow(batch_images[0].permute(1, 2, 0))
plt.show()
```

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

```python
from torch.utils.data import TensorDataset, DataLoader

# Wrap tensors in a Dataset, then DataLoader
train_dataset = TensorDataset(X_train, y_train)
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True) # 32 samples per batch, random

model_mb = LinearRegressionModel(input_dim=X_train.shape[1]).to(DEVICE)
criterion = nn.MSELoss()
optimizer_mb = optim.Adam(model_mb.parameters(), lr=0.01)

model_mb.train() # Training mode so dropout/BatchNorm (if any) behave correctly during training.

n_epochs = 500
for epoch in range(n_epochs):
    epoch_loss = 0.0
    for batch_x, batch_y in train_loader: # Batch = a subset of the training data. One forward/backwa
        batch_x = batch_x.to(DEVICE) # size (batch_size, n_features)
        batch_y = batch_y.to(DEVICE) # size (batch_size,)

        # Forward: predictions and loss for this batch only.
        pred = model_mb(batch_x)
        loss = criterion(pred, batch_y.unsqueeze(1))  # target shape (batch_size, 1) to match pred

        # Backward: compute gradients, then optimizer step (update parameters).
```

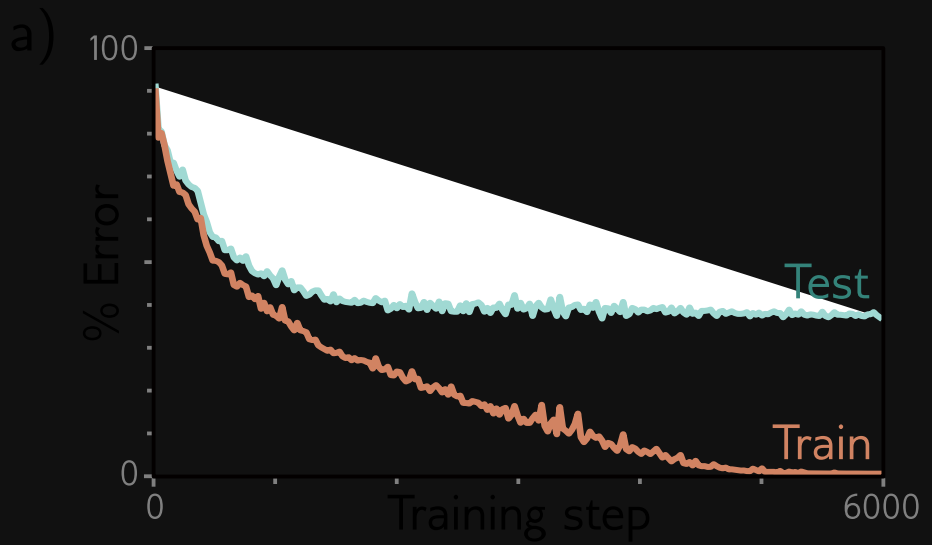
```
Epoch 100/500, mean loss: 28557.7874
Epoch 200/500, mean loss: 25439.1797
Epoch 300/500, mean loss: 23249.3605
Epoch 400/500, mean loss: 22399.3983
```

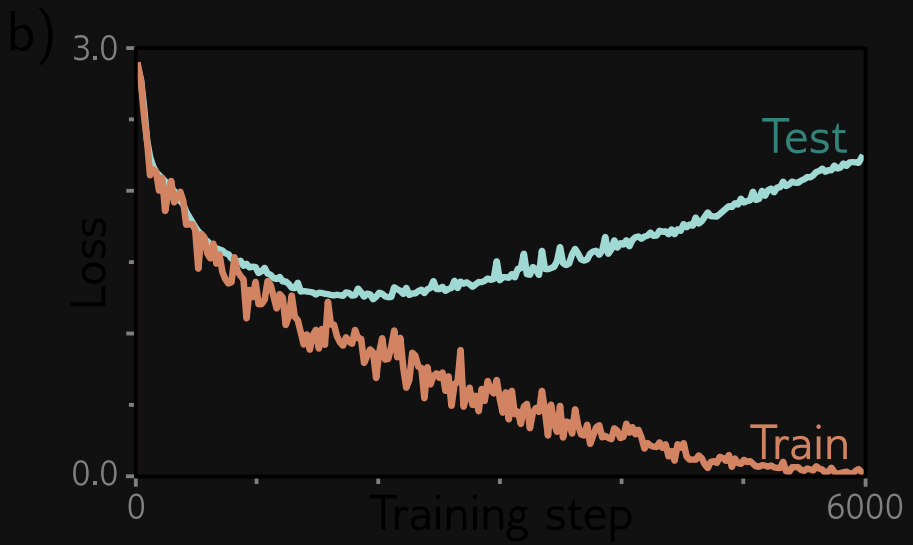
```
Epoch 500/500, mean loss: 21097.8613
Sample predictions (first 3): [22.177624 29.933098 28.115124]
```

Model Evaluation

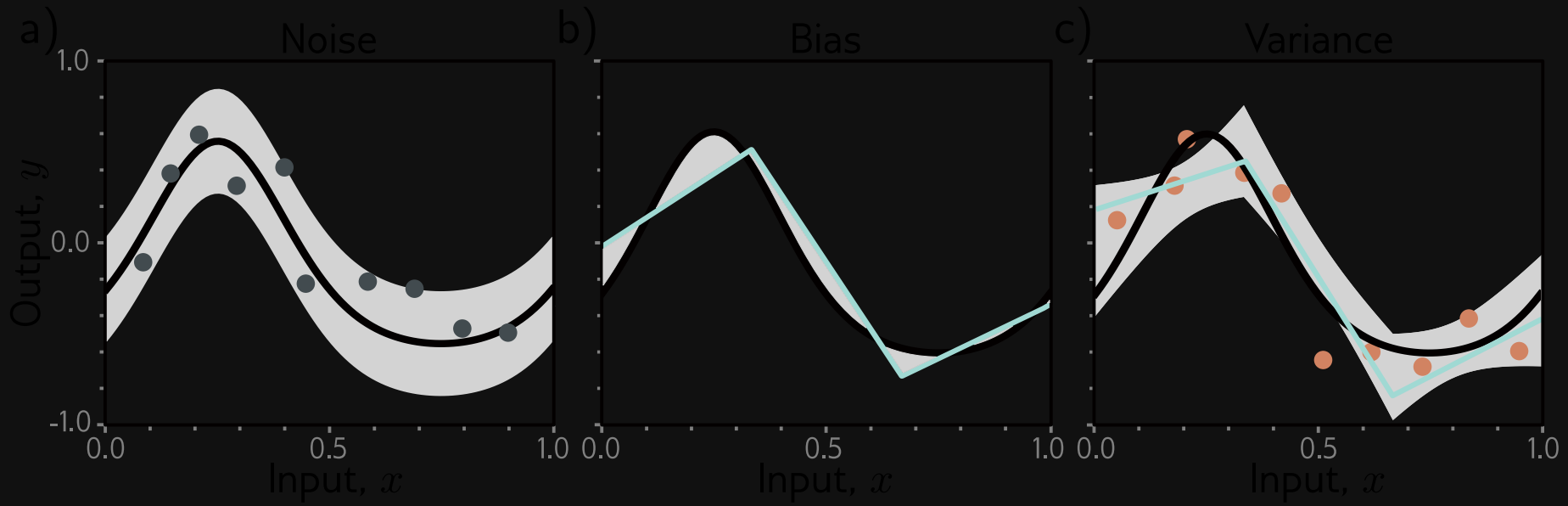
Prince (2023, chap. 8)

Training and Test Error





Error Sources

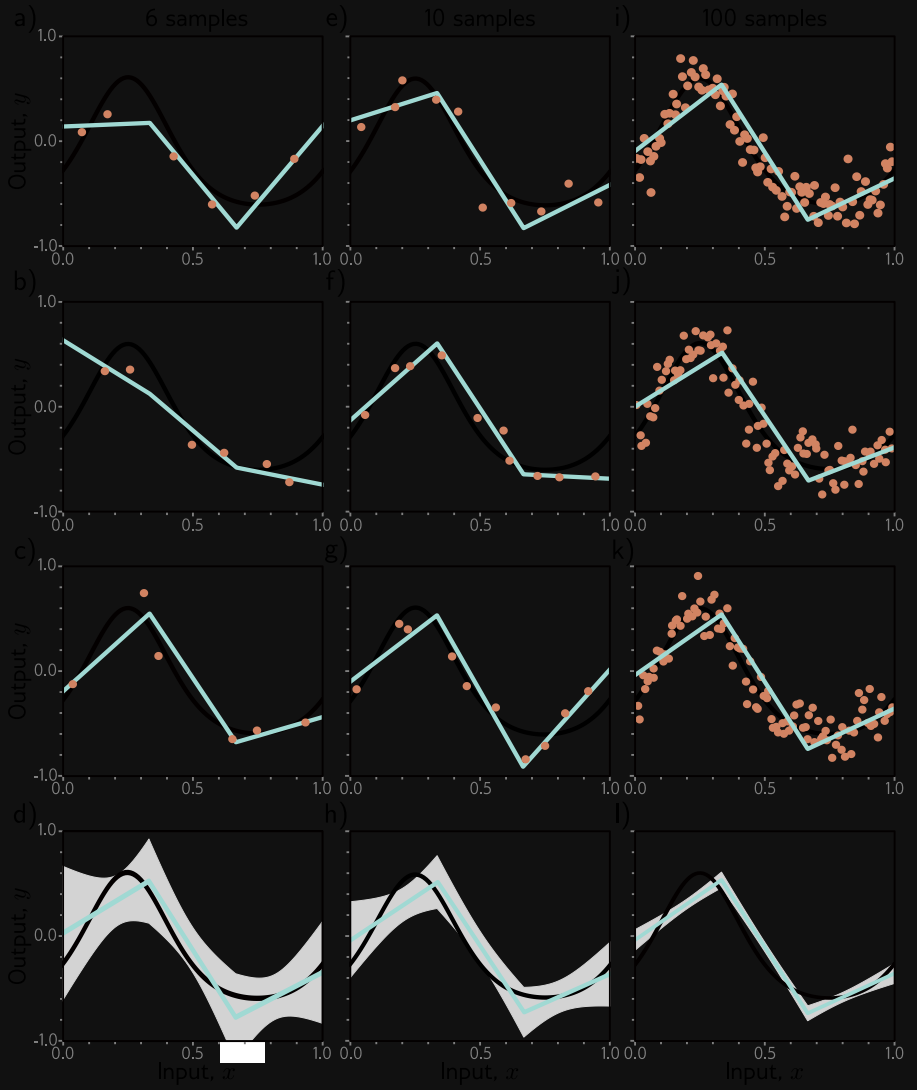


Error Decomposition

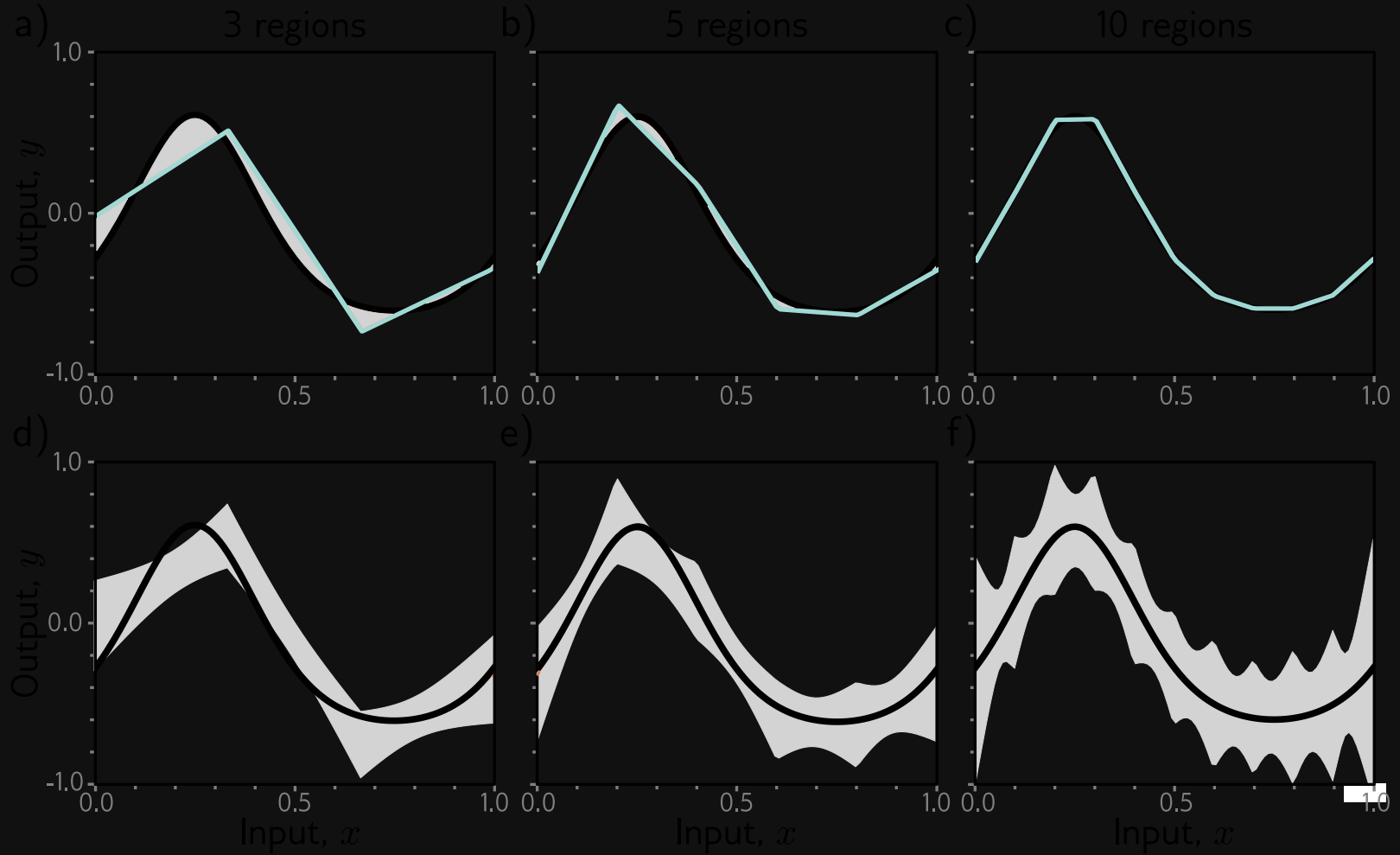
In a linear regression, and hence are stochastic, so taking expectations w.r.t. test data Taking expectations with respect to training data where

- variance due to sample noise in training data
- bias due to model misspecification
- noise due to inherent uncertainty in

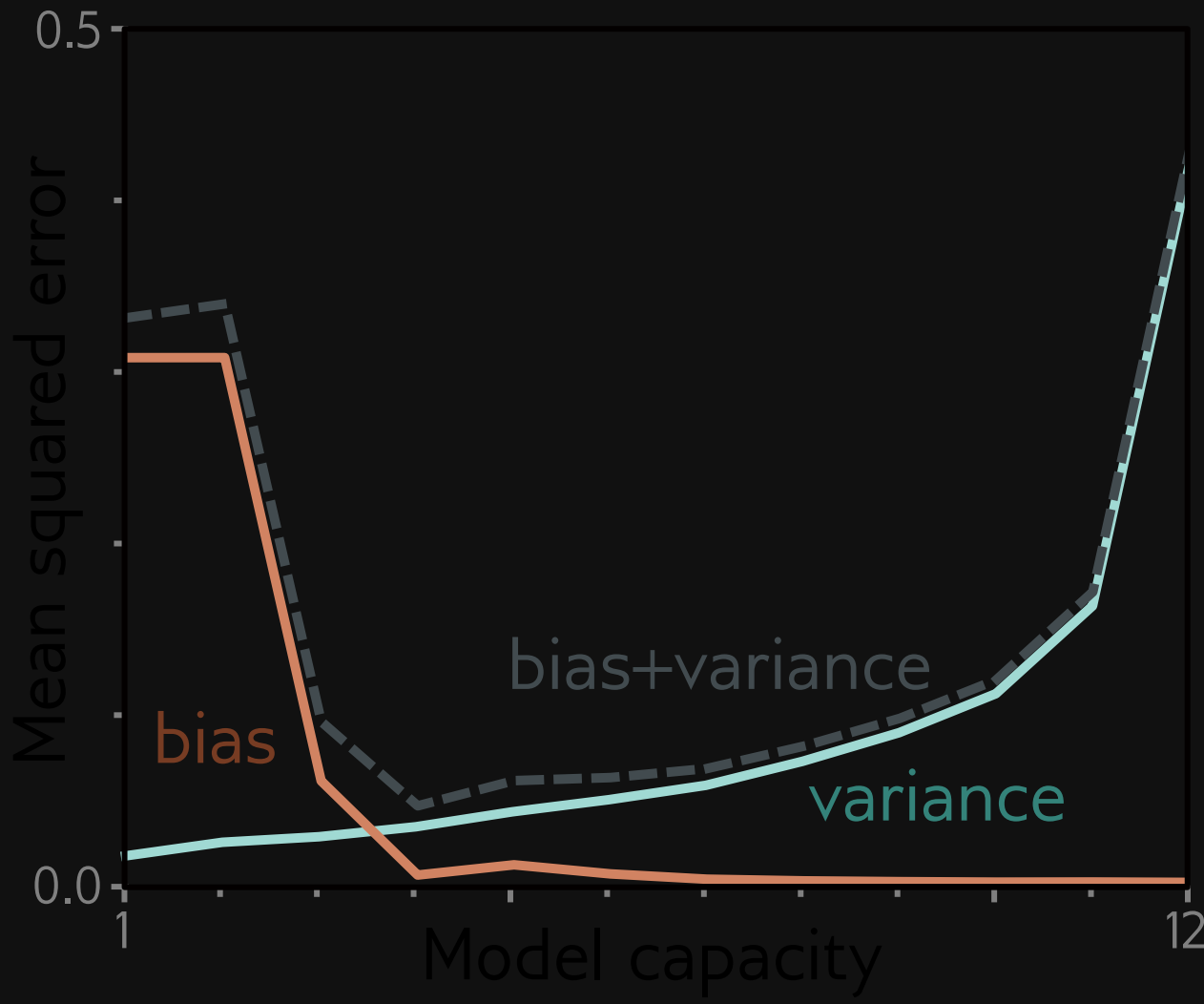
More Training Data: Less Variance



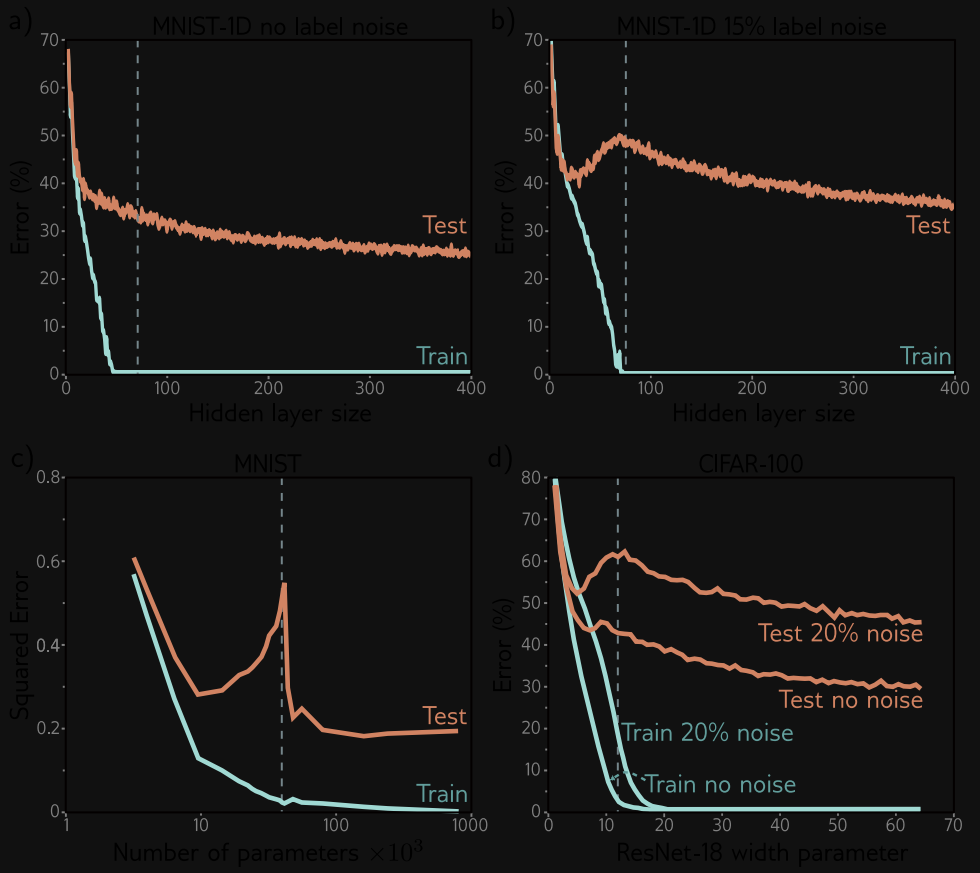
Increasing Model Capacity: Less Bias but More Variance



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

References

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