



CAMBRIDGE

# Lecture 1: Regression

Stefan Bucher

MACHINE LEARNING IN ECONOMICS  
UNIVERSITY OF CAMBRIDGE

Stefan Bucher

 Open in Colab

## Prince (2023, chaps. 2, 6, 8).<sup>1</sup>

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

# Linear Regression

Prince (2023, chap. 2)

# Linear Model

$$\mathbf{y} = \mathbf{X}\beta + \varepsilon$$

Minimizing the mean-squared error (MSE)

$$\min_{\beta} \frac{1}{n} (\mathbf{y} - \mathbf{X}\beta)^T (\mathbf{y} - \mathbf{X}\beta)$$

# Minimizing MSE Loss (2.1-2.3)

# Analytical Solution

$$\min_{\beta} \frac{1}{n} (y - X\beta)^T (y - X\beta)$$

Setting the gradient with respect to  $\beta$  to zero

$$0 = \nabla_{\beta} [y^T y - 2\beta^T X^T y + \beta^T X^T X\beta] = -2X^T y + 2X^T X\beta$$

yields the BLUE estimator (Gauss-Markov)

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

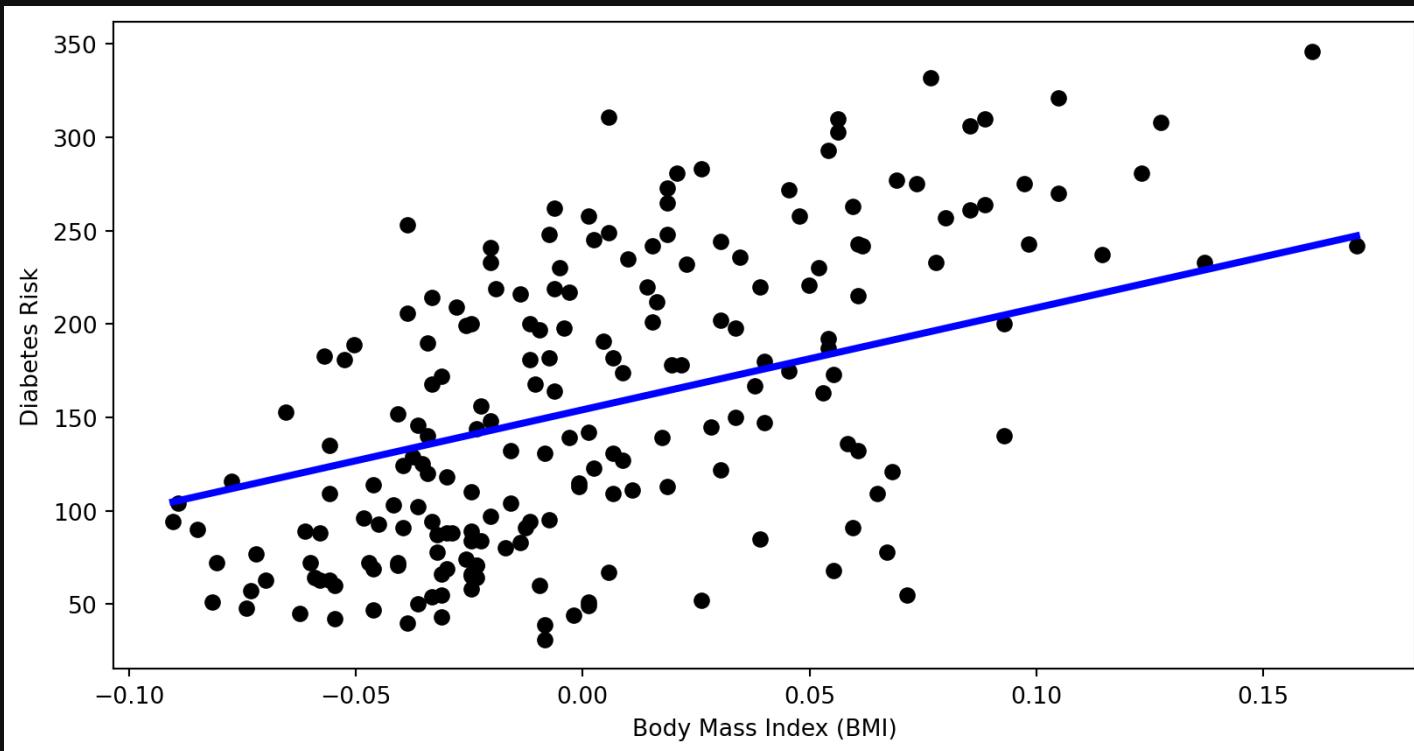
# scikit-learn

# scikit-learn

```
1 from sklearn.linear_model import LinearRegression  
2 model = LinearRegression()  
3 model.fit(X_train, y_train)  
4 print(model.coef_[2], model.intercept_)
```

546.1898661888245 152.05949984245817

Text(0, 0.5, 'Diabetes Risk')



# Model Fitting & Optimization

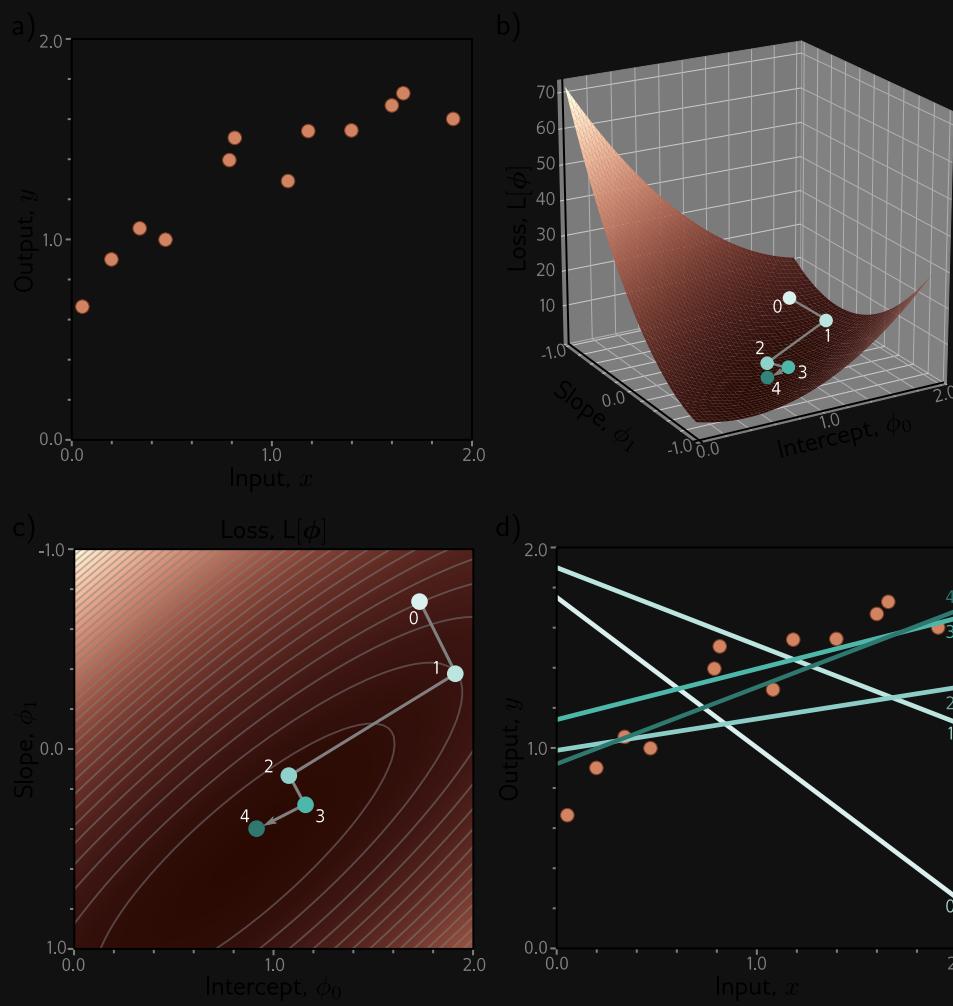
$$\hat{\phi} = \arg \min_{\phi} L(\phi)$$

Prince (2023, chap. 6)

# Gradient Descent<sup>1</sup>

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

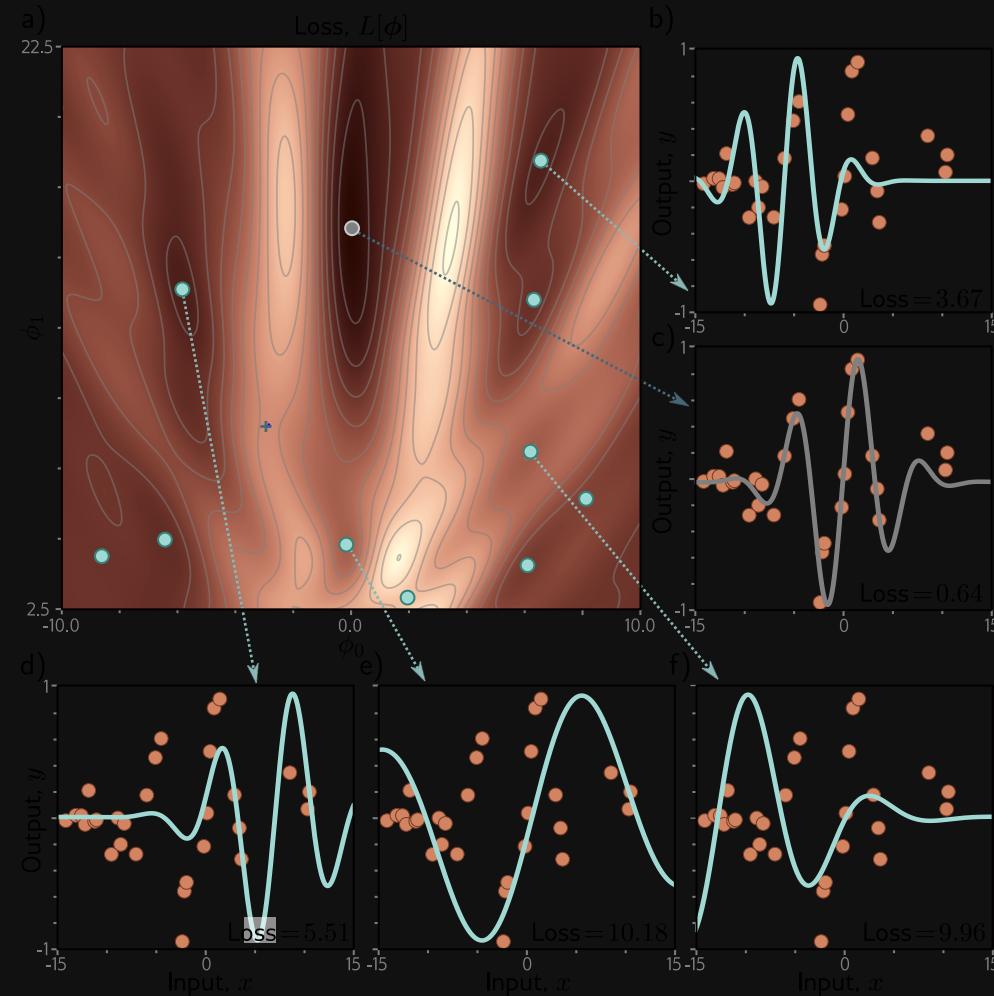
where  $\alpha$  fixed learning rate or line search.



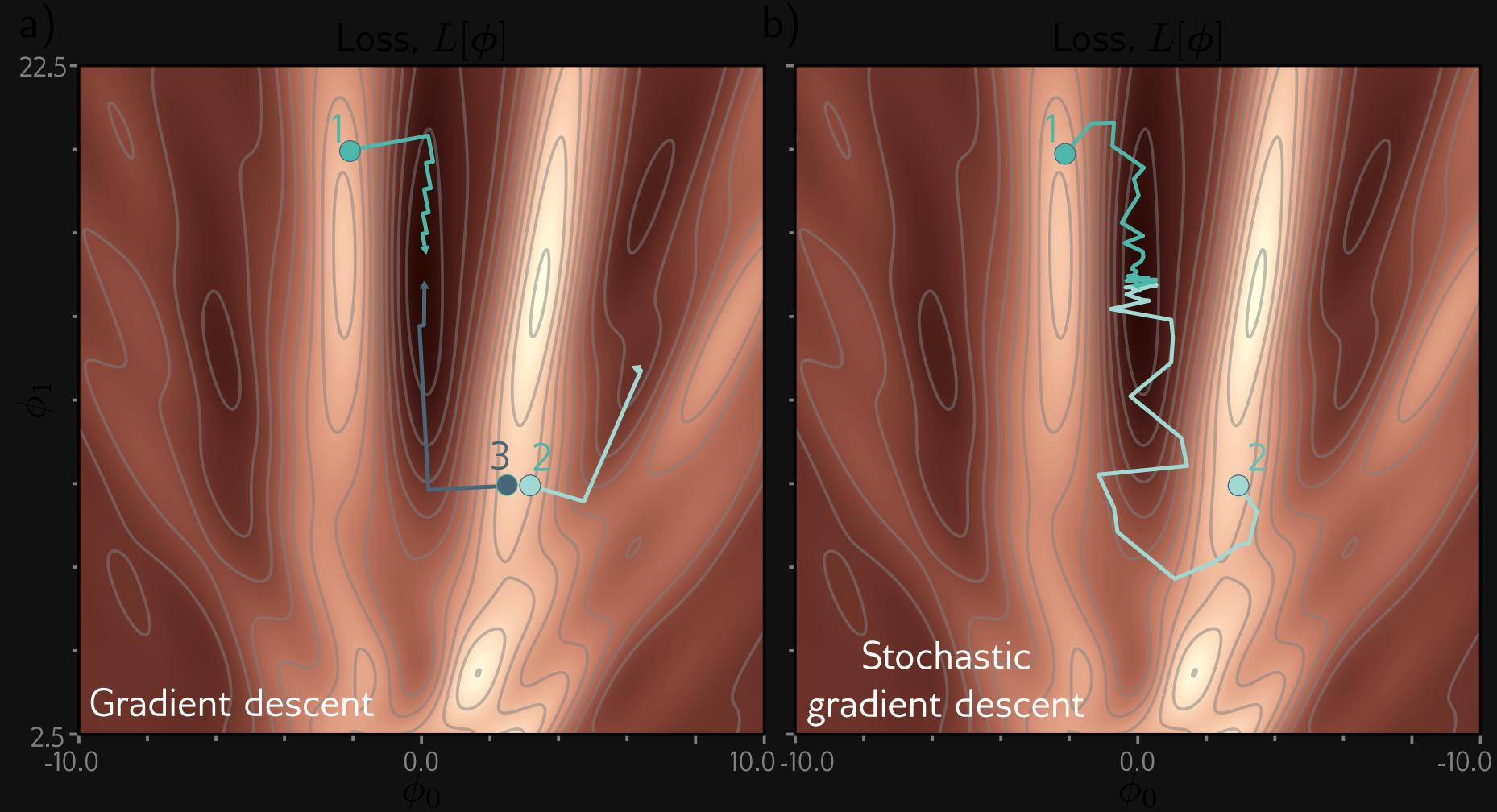
• Cauchy (1847)

# Nonlinear Model

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



# Stochastic Gradient Descent

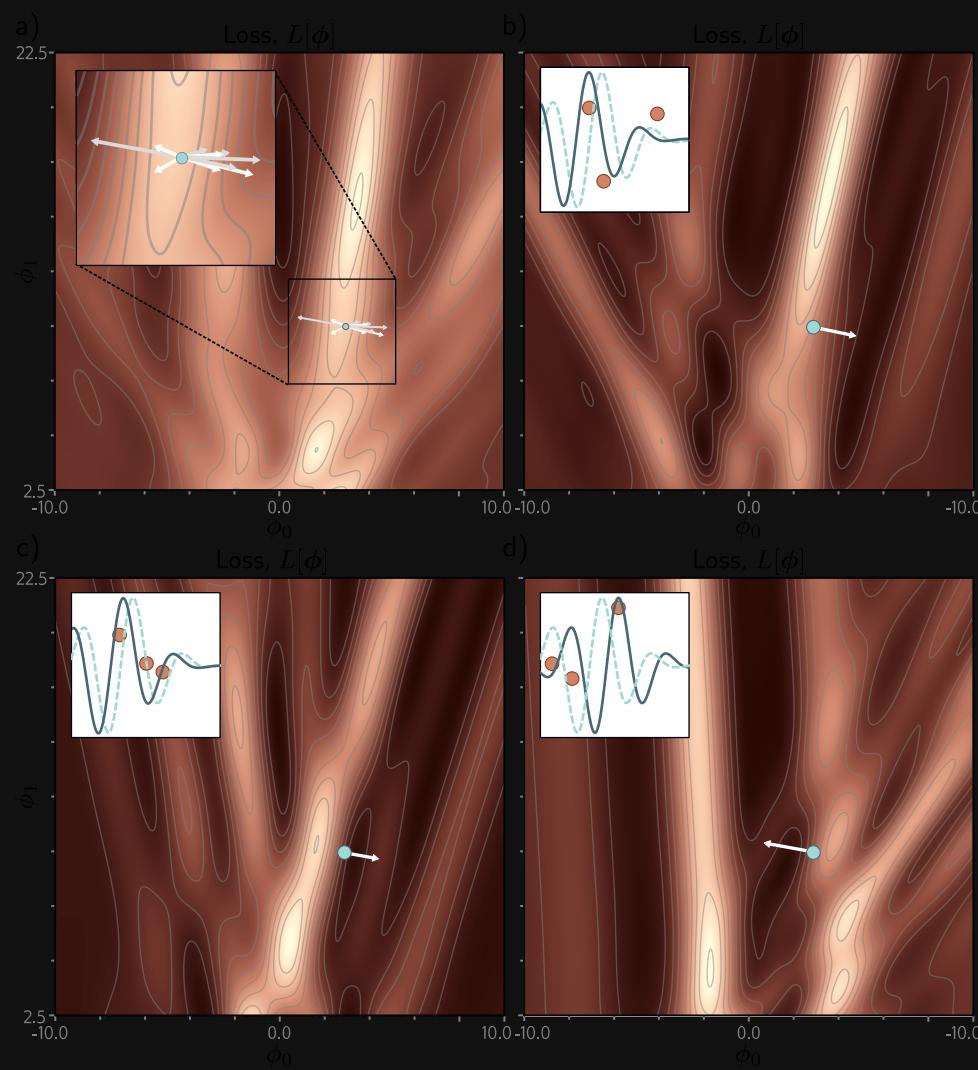


# Stochastic Gradient Descent<sup>1</sup>

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.



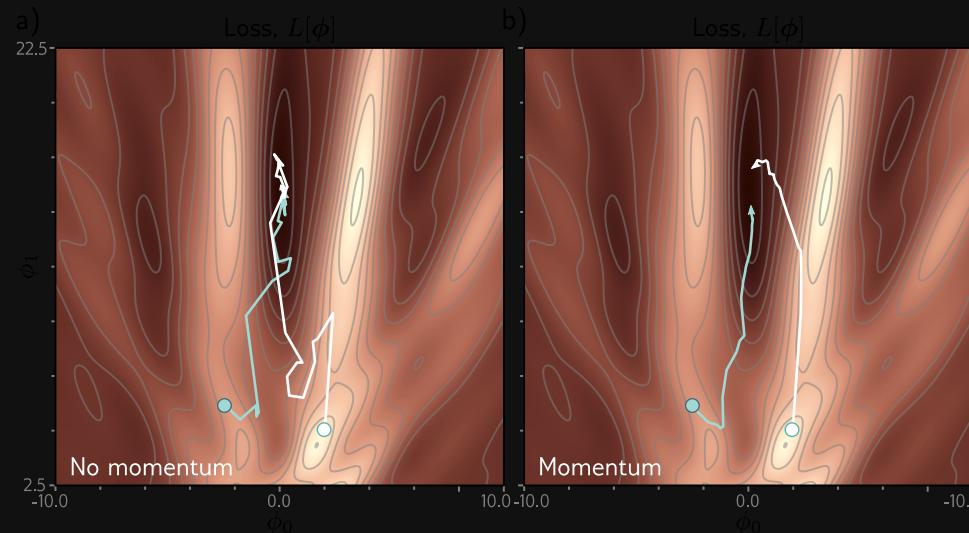
. Robbins & Monro (1951)

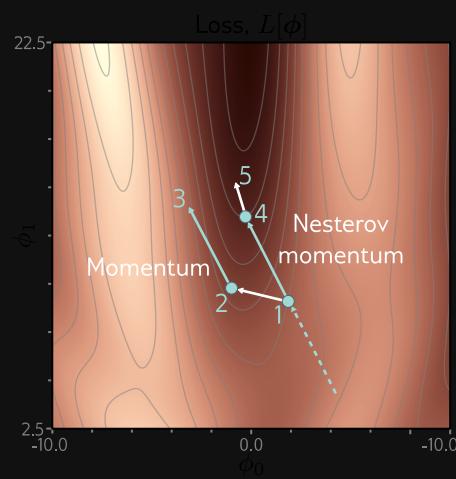
# Nesterov Accelerated Momentum<sup>1</sup>

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

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

adds momentum ( $m_t$ ) *before* evaluating gradient.





. Nesterov (1983)

# Adaptive Moment Estimation (Adam)<sup>1</sup>

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

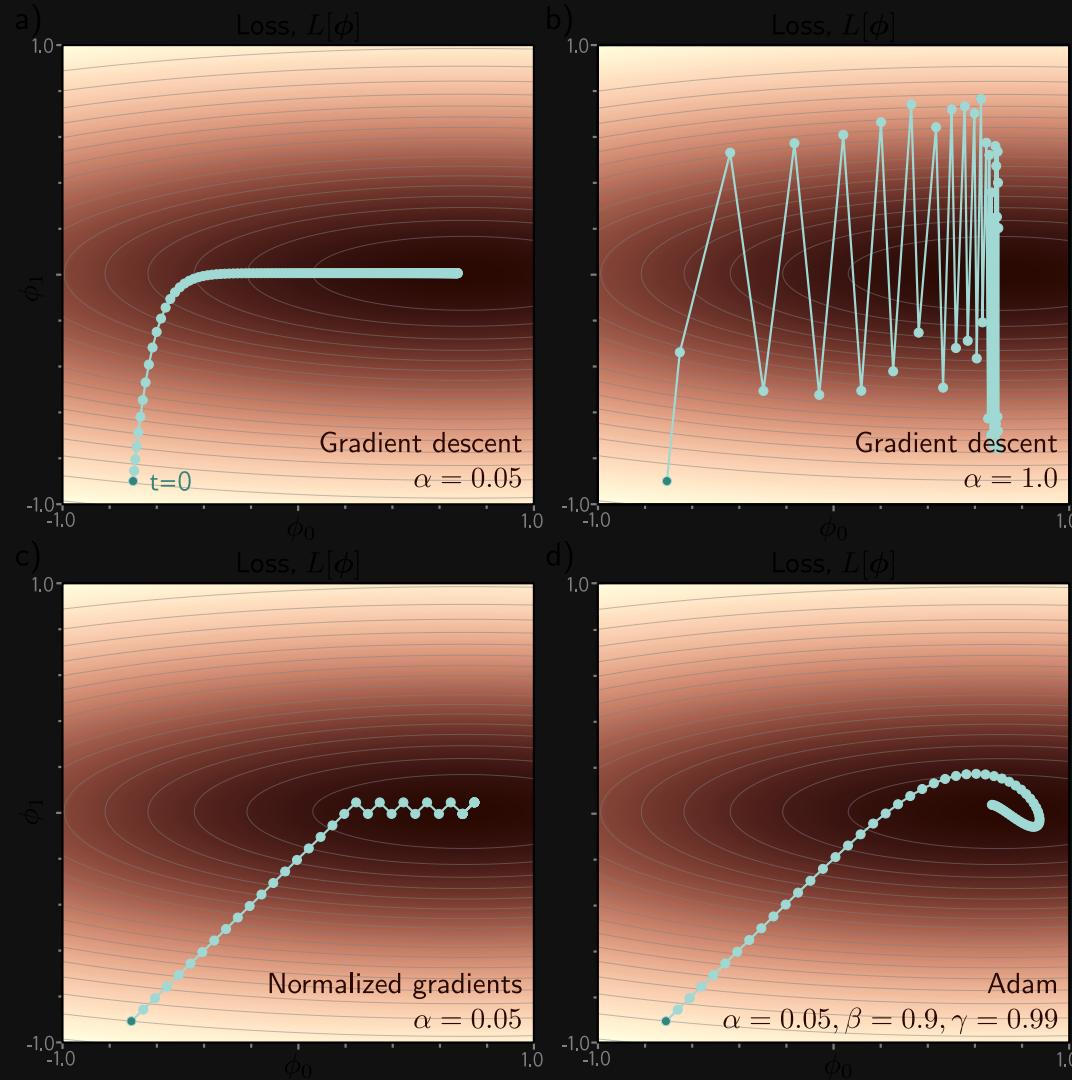
$$m_{t+1} \leftarrow \beta m_t + (1 - \beta) \sum_{i \in B_t} \nabla_{\phi} l_i(\phi_t; x_i, y_i)$$

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

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

. Kingma & Ba (2015)

# Adaptive Moment Estimation (Adam)



# PyTorch

# PyTorch

# Setup

```
1 import torch
2 from torch import nn
3 import torch.optim as optim
4 from torchvision import datasets
5 from torch.utils.data import DataLoader
6 from torchvision.transforms import ToTensor, Compose, Grayscale
7 import random
8
9 if torch.backends.mps.is_available():
10     DEVICE = torch.device("mps") # Apple Silicon Metal Performance Shaders
11 elif torch.cuda.is_available():
12     DEVICE = torch.device("cuda") # GPU
13 else:
14     DEVICE = torch.device("cpu") # CPU
```

## Key components:

- tensors are like numpy arrays but GPU-accelerated
- autograd

# Tensors

Tensors “live” on a device

```
1 X_train = torch.tensor(X.values, dtype=torch.float32).to(DEVICE)
2 y_train = torch.tensor(y.values, dtype=torch.float32).to("cpu")
3 print(X_train.device)
4 X_train = X_train.to("cpu")
5 print(X_train.device)
```

mps:0  
cpu

# The Previous Regression

```
1 solution = torch.linalg.lstsq(X_train, y_train)
2 coefficients = solution.solution[:-1].flatten() # All coefficients except the intercept
3 intercept = solution.solution[-1].item() # Intercept
4 print(coefficients[2], intercept)
```

```
tensor(519.8459) 152.13348388671875
```

# The Previous Regression

```
1 class LinearRegressionModel(nn.Module):
2     def __init__(self, input_dim):
3         super(LinearRegressionModel, self).__init__()
4         self.linear = nn.Linear(input_dim, 1)
5
6     def forward(self, x):
7         return self.linear(x)
8
9 model = LinearRegressionModel(input_dim=X_train.shape[1])
10 criterion = nn.MSELoss()
11 # optimizer = optim.SGD(model.parameters(), lr=0.01)
12 optimizer = optim.Adam(model.parameters(), lr=0.01)
13
14 # Training loop
15 epochs = 15000
16 for epoch in range(epochs):
17     # Forward pass
18     outputs = model(X_train)
19     loss = criterion(outputs, y_train)
20
21     # Backward pass and optimization
22     optimizer.zero_grad()
23     loss.backward()
24     optimizer.step()
```

Epoch [1000/15000], Loss: 23484.2832

Epoch [2000/15000], Loss: 18946.1602

Epoch [3000/15000], Loss: 15241.5107

Epoch [4000/15000], Loss: 12269.5684

Epoch [5000/15000], Loss: 9959.8135

Epoch [6000/15000], Loss: 8253.8252

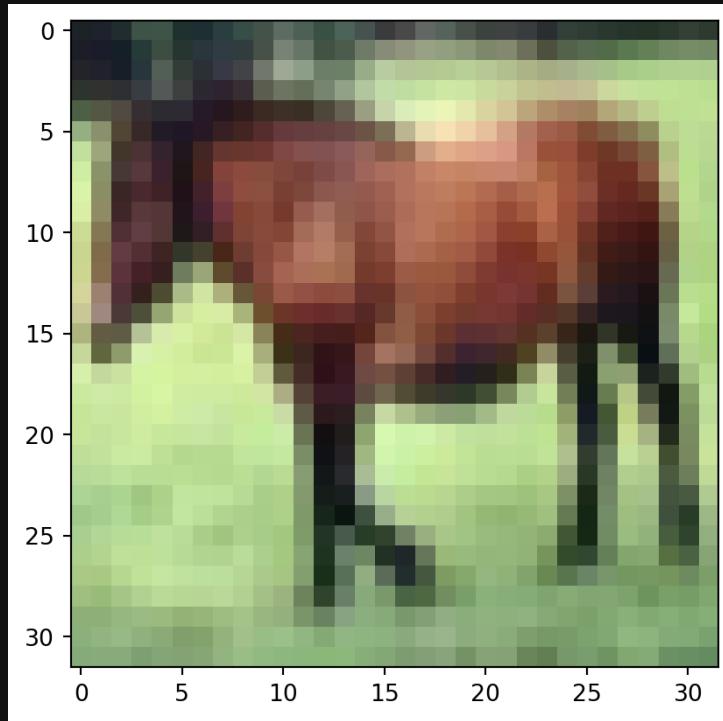
```
Epoch [7000/15000], Loss: 7090.5425
Epoch [8000/15000], Loss: 6393.1763
Epoch [9000/15000], Loss: 6057.3301
Epoch [10000/15000], Loss: 5948.0991
Epoch [11000/15000], Loss: 5930.6899
Epoch [12000/15000], Loss: 5929.8892
Epoch [13000/15000], Loss: 5929.8848
Epoch [14000/15000], Loss: 5929.8848
Epoch [15000/15000], Loss: 5929.8848
linear.weight: [[ 3.61002094e-05 -1.17296531e-05 -2.93517442e-05  4.17710726e-06
   1.18487769e-04 -1.26960615e-04 -4.82288597e-05  1.02682825e-05
  -5.74503320e-05 -2.99520252e-07  7.60413208e+01]]
linear.bias: [76.09164]
```

# Datasets

```
1 training_data = datasets.CIFAR10(  
2     root="data", # path where the images will be stored  
3     train=True,  
4     download=True,  
5     transform=ToTensor()  
6     )  
7 test_data = datasets.CIFAR10(  
8     root="data",  
9     train=False,  
10    download=True,  
11    transform=ToTensor()  
12    )  
13 image, label = training_data[7]  
14 print(f"Label: {training_data.classes[label]}")  
15 print(f"Image size: {image.shape}")  
16 plt.imshow(image.permute(1, 2, 0)) # imshow expects channel order Height x Width x Color(RGB)  
17 plt.show()
```

Label: horse

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



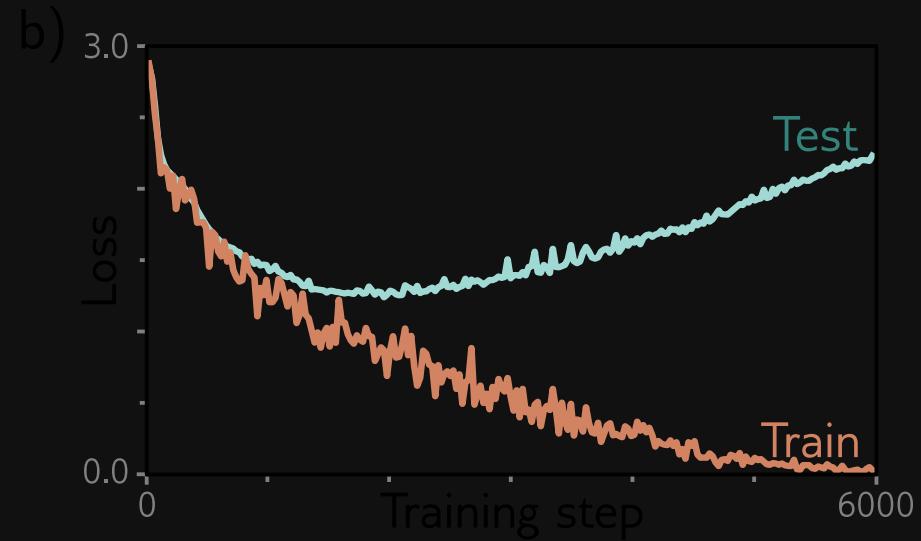
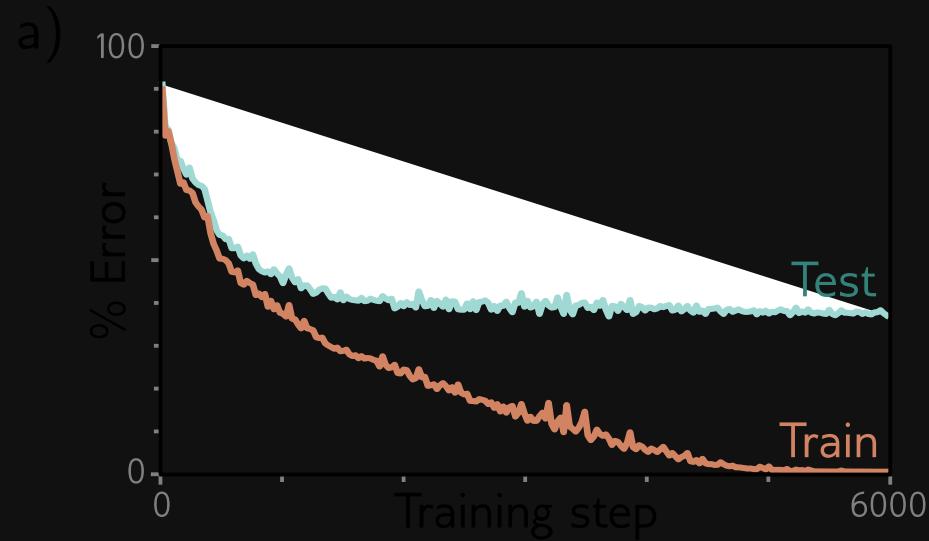
# Data Loader

```
1 # Data Loader for Batching
2 def seed_worker(worker_id):
3     worker_seed = torch.initial_seed() % 2**32
4     numpy.random.seed(worker_seed)
5     random.seed(worker_seed)
6     g_seed = torch.Generator()
7     g_seed.manual_seed(torch.initial_seed() % 2**32)
8
9 train_dataloader = DataLoader(training_data, batch_size=64, shuffle=True, num_workers=2, worker_init_
10 test_dataloader = DataLoader(test_data,  batch_size=64, shuffle=True, num_workers=2, worker_init_fn=s
11
12 # Load the next batch
13 batch_images, batch_labels = next(iter(train_dataloader))
14 print('Batch size:', batch_images.shape)
15 plt.imshow(batch_images[0].permute(1, 2, 0))
16 plt.show()
```

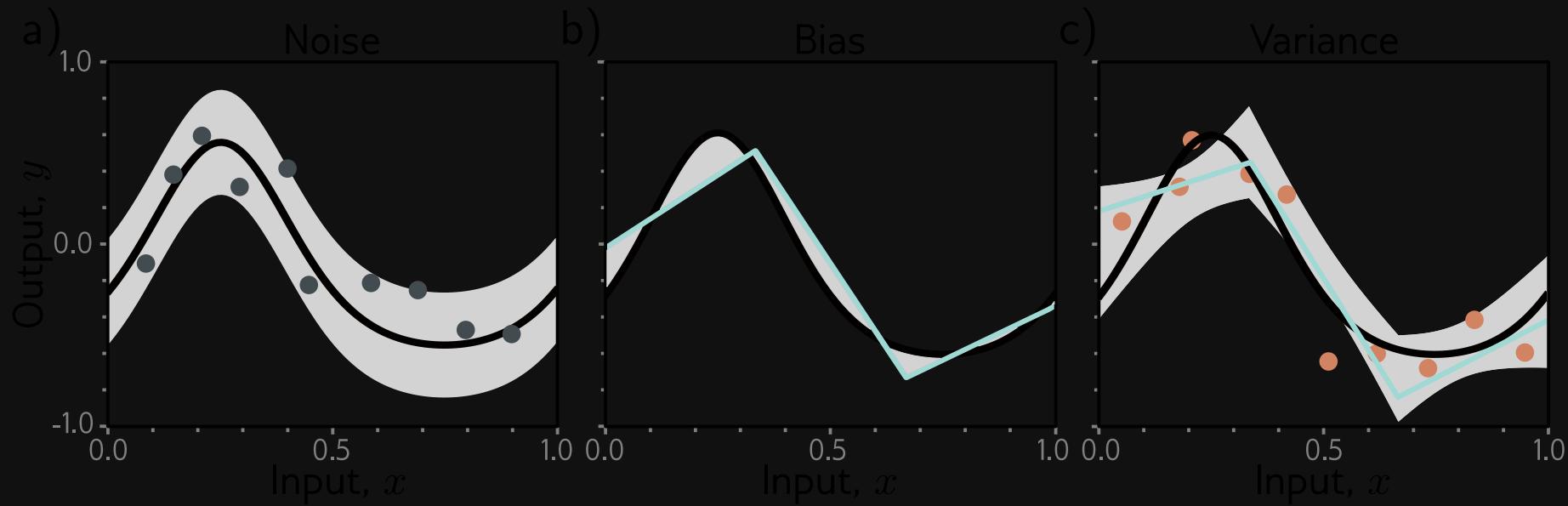
# Measuring Performance

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  $y$

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

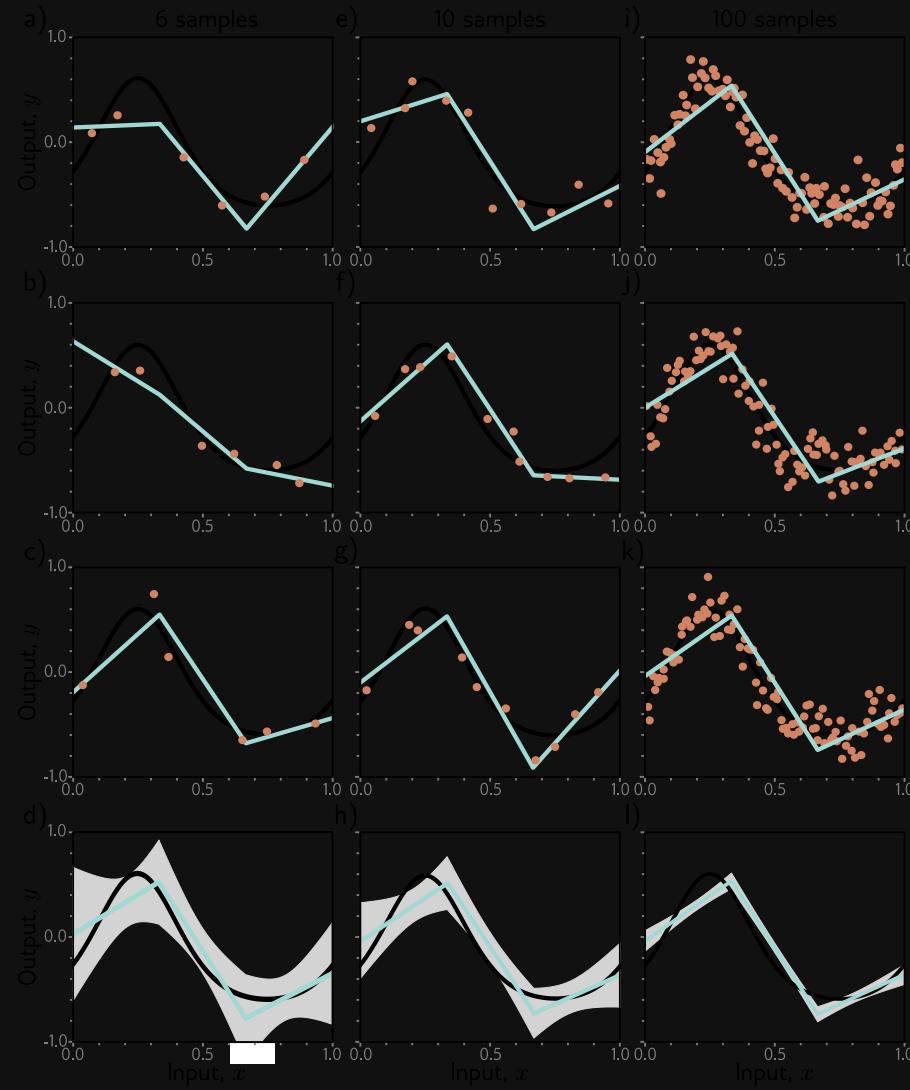
Taking expectations with respect to training data  $D$

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

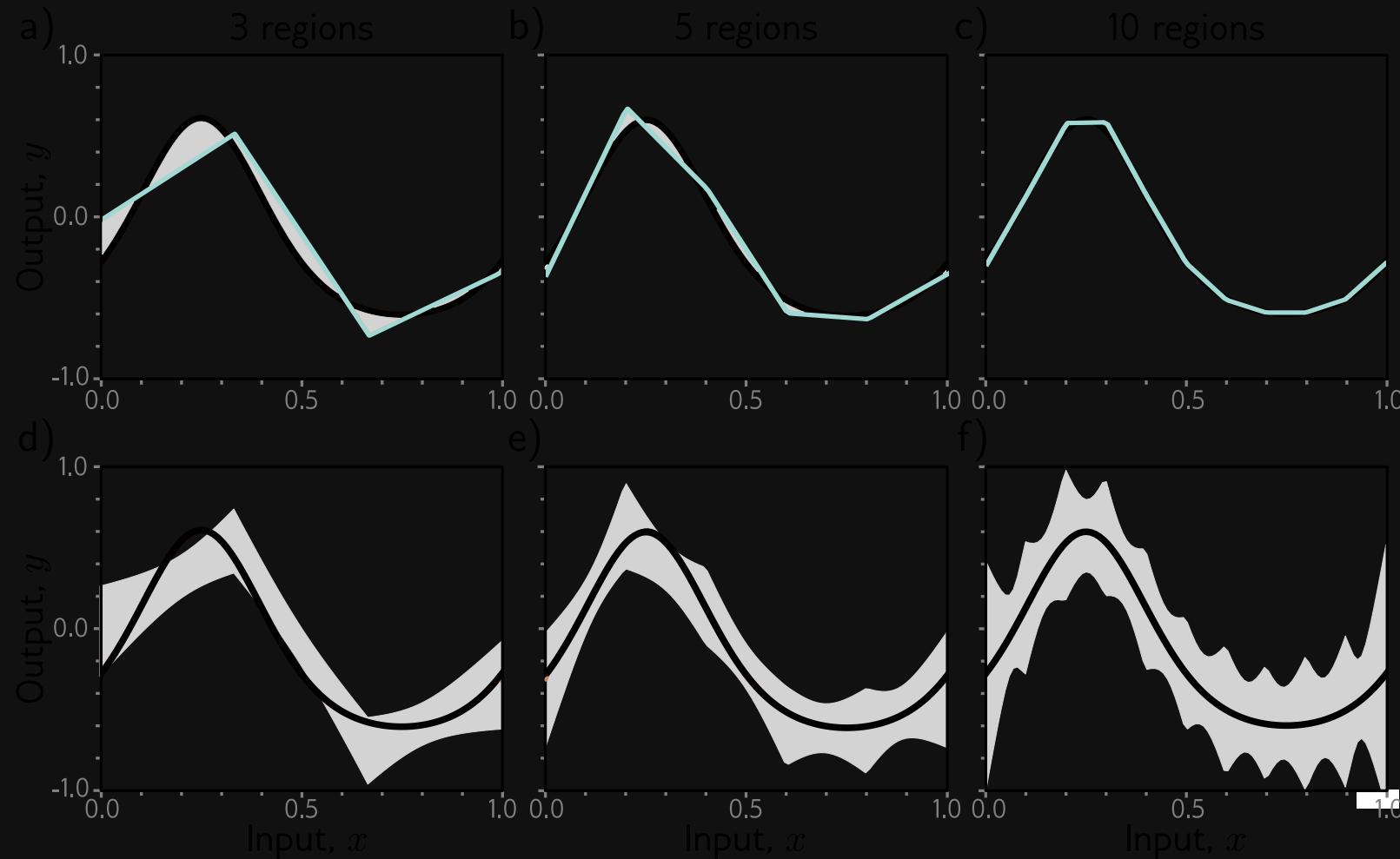
where

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

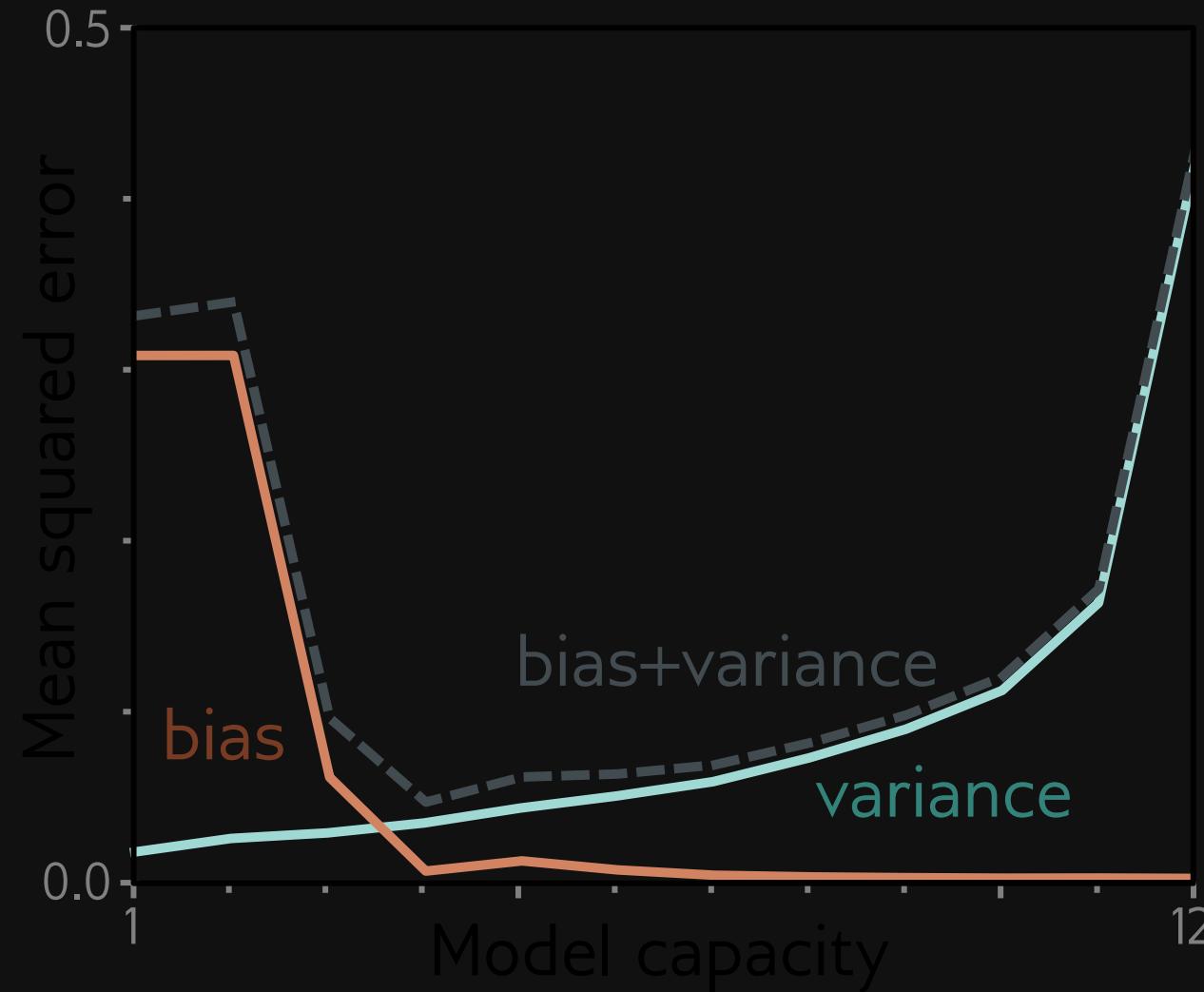
# More Training Data: Less Variance



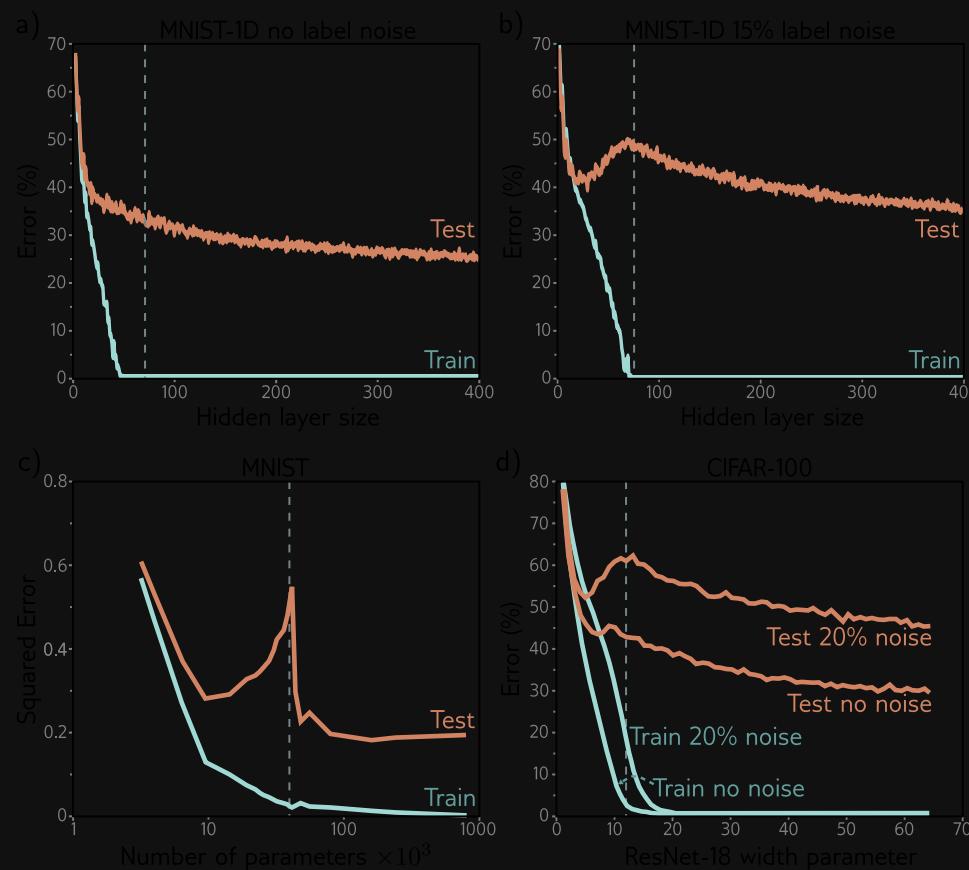
# Increasing Model Capacity: Less Bias but More Variance



# 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

# References

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