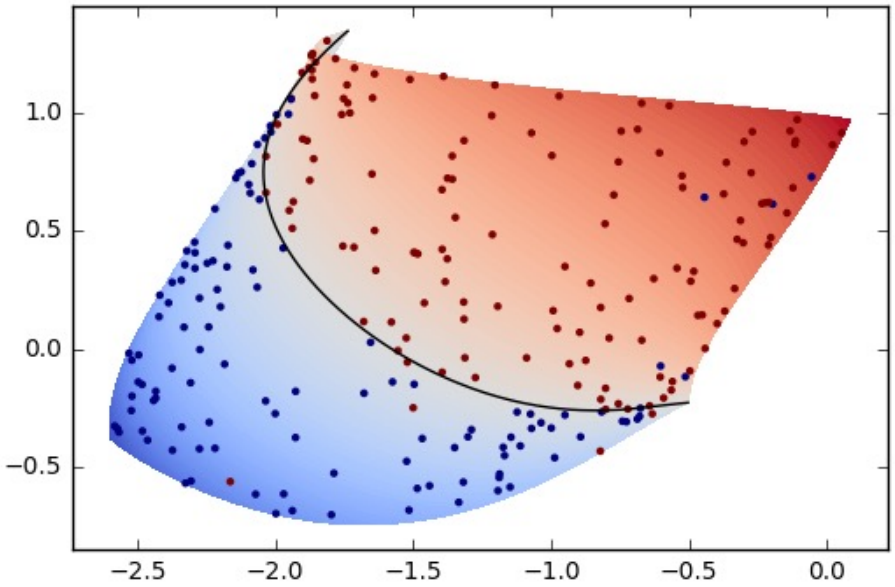
Lecture 16: Neural Networks Part 4



Gavin Kerrigan

Spring 2023

Some materials courtesy Padhraic Smyth, Alex Ihler.

Announcements

- Homework 3 – due Friday
  - Part of today's lecture will take a look at this assignment
- Projects
  - Team formation due in 2 weeks (Friday 5/19)
- Midterm exam
  - Grading in progress
  - Maybe a little long
- Course Survey
  - 133/162 students responded
  - Thank you! Your feedback is greatly appreciated

Lectures

Attempts: 133 out of 133

The explanations of concepts during lecture are clear.

Strongly Disagree	5 respondents	4 %
Disagree	7 respondents	5 %
Agree	71 respondents	53 %
Strongly Agree	50 respondents	38 %

Attempts: 133 out of 133

The pacing of the lectures is...

Too slow	2 respondents	2 %
Just right	99 respondents	74 %
Too fast	32 respondents	24 %

Attempts: 133 out of 133

With regards to answering questions from students during lecture, the instructor should spend...

More time	15 respondents	11 %
The same amount of time	109 respondents	82 %
Less time	9 respondents	7 %

Homeworks

Attempts: 133 out of 133

The homeworks are sufficiently well-integrated with the lectures.

Strongly Disagree	4 respondents	3 %
Disagree	22 respondents	17 %
Agree	82 respondents	62 %
Strongly Agree	25 respondents	19 %

Attempts: 133 out of 133

The questions in the homework assignments are sufficiently concrete and clear.

Strongly Disagree	4 respondents	3 %
Disagree	26 respondents	20 %
Agree	81 respondents	61 %
Strongly Agree	22 respondents	17 %

Discussions, Probability, Math

Attempts: 133 out of 133

The discussion sections have been helpful in demonstrating the concepts covered in lecture.

Strongly Disagree	4 respondents	3 %
Disagree	28 respondents	21 %
Agree	84 respondents	63 %
Strongly Agree	17 respondents	13 %

Attempts: 133 out of 133

The review lecture on probability was too basic, and I would have preferred that we use this lecture to cover more machine learning topics rather than review probability.

Strongly Disagree	11 respondents	8 %
Disagree	67 respondents	50 %
Agree	47 respondents	35 %
Strongly Agree	8 respondents	6 %

Attempts: 133 out of 133

The balance between mathematical/theoretical content and practical/hands-on content is appropriate in lecture.

I'd prefer more math	4 respondents	3 %
Just right	69 respondents	52 %
I'd prefer more practical details, e.g. code examples	60 respondents	45 %

Common suggestions/comments in free response:

- More hands-on coding examples in lecture
- More connections between theory (lecture) and practice (homework)
- Slower pacing on sections which are heavy on the math
- Homeworks are documentation “scavenger hunts”
    - Pretty good simulation of what ”real-world” machine learning is like!

Neural Network Recap

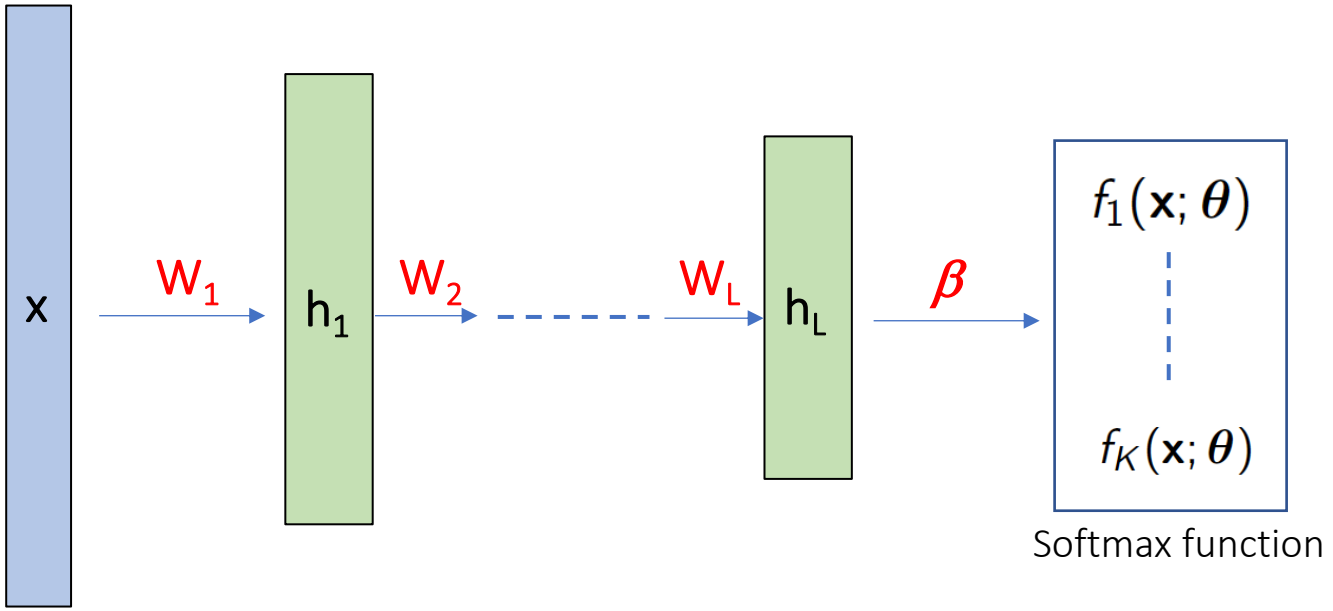
Neural Networks in Code

Tuning a Network

Summary of Neural Networks

- Neural network = non-linear function $f(x; \theta)$
- Parameters θ = weight matrices and bias terms
- Feedforward neural network
  - Layers of weight matrices + hidden unit non-linearities
  - Non-linearities => non-linear classification boundaries
    - ReLU or logistic used as hidden unit non-linearities
  - Softmax used at the output for K-way classification
- Deep neural networks
  - Used to describe neural networks with multiple hidden layers

General Feedforward Neural Network for Classification



This is for K > 2 classes.
With 2 classes, just a single (sigmoid) output

Known as a **feedforward fully-connected neural network** (also: "multilayer perceptron")

Fully-connected means all hidden units are connected to all hidden units at the next layer; also referred to as "**dense**"

Example of a **fully connected dense neural network** with 7 inputs, 3 hidden layers, 12 outputs

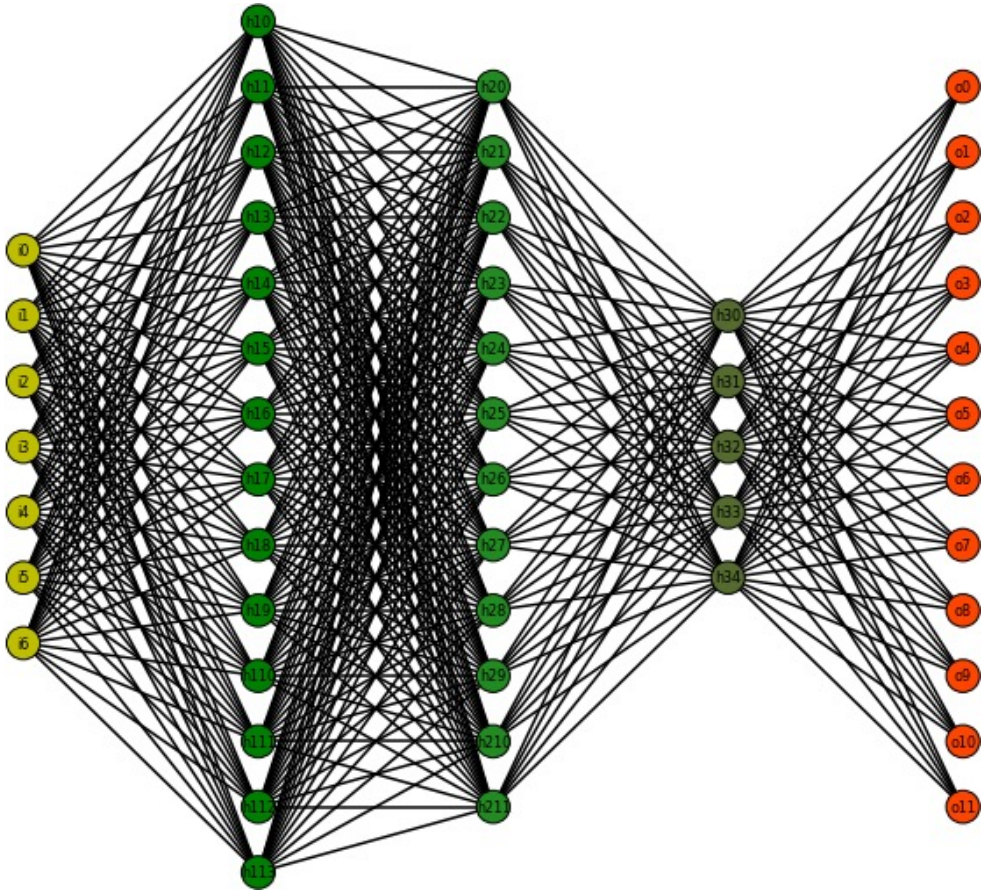


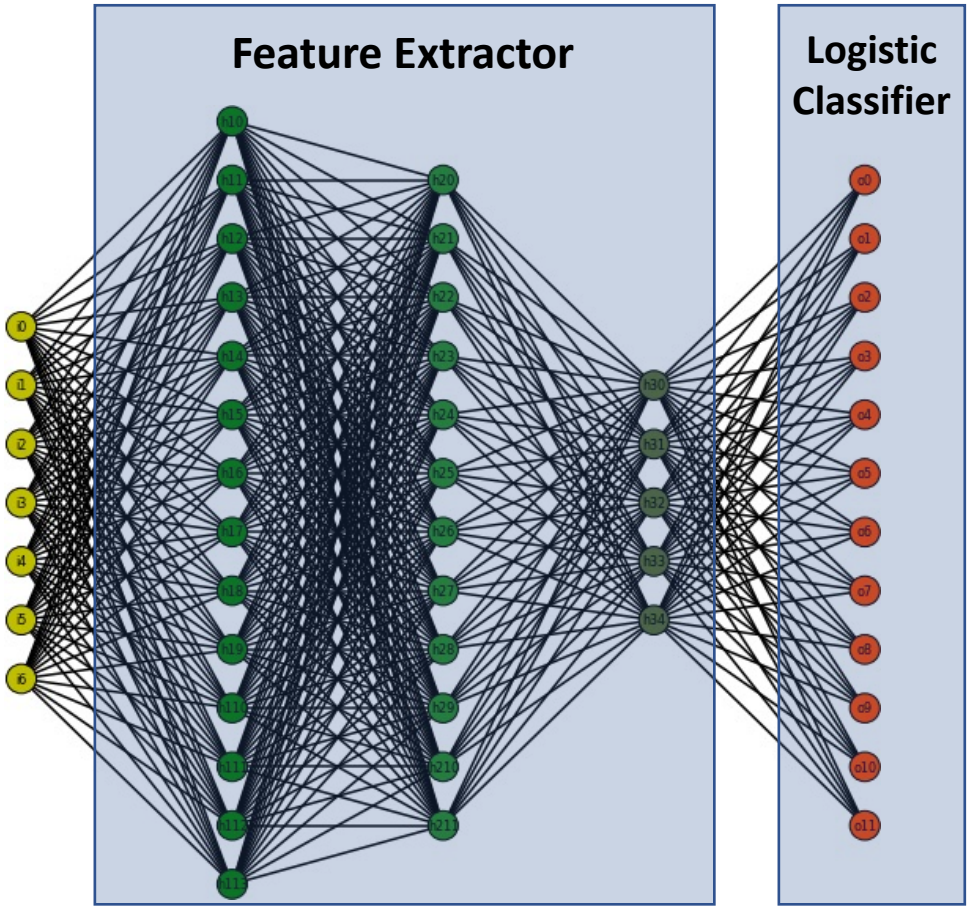
Figure from https://www.bodenseo.de/kurs/machine_learning.html

Summary of Neural Networks (2)

- Neural networks are trained by gradient descent
  - Backpropagation used to update parameters θ based on gradient information "propagated" from output to earlier layers
- Stochastic gradient descent
  - Allows parameters to be updated more often by "noisy gradient"
  - Can converge much more quickly than standard gradient on large data sets
- Neural networks work very well when
  - large amount of training data
  - feature extraction from low-level signals is important
    - E.g., vision, speech, text,…

Representation Learning

Hidden units at each layer in a neural network can be thought of as learning useful features (or “representations”)



Questions?

Neural Network Recap

Neural Networks in Code

Tuning a Network

```python
class NeuralNetwork:
    def __init__(self, input_dim, h1_dim, h2_dim, n_classes):

        # Weights and biases in first hidden layer
        self.b1 = np.random.randn()
        self.W1 = np.random.randn(h1_dim, input_dim)

        # Weights and biases in second hidden layer
        self.b2 = np.random.randn()
        self.W2 = np.random.randn(h2_dim, h1_dim)

        # Weights and biases in output layer
        self.b3 = np.random.randn()
        self.beta = np.random.randn(n_classes, h2_dim)

    def forward(self, x):
        # x is a feature vector of shape (d,)

        h1 = np.matmul(self.W1, x) + self.b1      # "pre-activations" for first hidden layer
        h1 = self.relu(h1)                        # values of first hidden layer

        h2 = np.matmul(self.W2, h1) + self.b2     # "pre-activations" for second hidden layer
        h2 = self.relu(h2)                        # values of second hidden layer

        out = np.matmul(self.beta, h2) + self.b3  # "scores" before softmax of output
        out = self.softmax(out)

        return out
```

Simple neural network (2 hidden layers) in <30 lines of code

```python
class NeuralNetwork:
    def __init__(self, input_dim, h1_dim, h2_dim, n_classes):

        # Weights and biases in first hidden layer
        self.b1 = np.random.randn()
        self.W1 = np.random.randn(h1_dim, input_dim)

        # Weights and biases in second hidden layer
        self.b2 = np.random.randn()
        self.W2 = np.random.randn(h2_dim, h1_dim)

        # Weights and biases in output layer
        self.b3 = np.random.randn()
        self.beta = np.random.randn(n_classes, h2_dim)

    def forward(self, x):
        # x is a feature vector of shape (d,)

        h1 = np.matmul(self.W1, x) + self.b1      # "pre-activations" for first hidden layer
        h1 = self.relu(h1)                        # values of first hidden layer

        h2 = np.matmul(self.W2, h1) + self.b2     # "pre-activations" for second hidden layer
        h2 = self.relu(h2)                        # values of second hidden layer

        out = np.matmul(self.beta, h2) + self.b3  # "scores" before softmax of output
        out = self.softmax(out)

        return out
```

Specify:

- Input dimension (# of features)
- Size of hidden layers
- Output dimension (# of classes)

```python
class NeuralNetwork:
    def __init__(self, input_dim, h1_dim, h2_dim, n_classes):

        # Weights and biases in first hidden layer
        self.b1 = np.random.randn()
        self.W1 = np.random.randn(h1_dim, input_dim)

        # Weights and biases in second hidden layer
        self.b2 = np.random.randn()
        self.W2 = np.random.randn(h2_dim, h1_dim)

        # Weights and biases in output layer
        self.b3 = np.random.randn()
        self.beta = np.random.randn(n_classes, h2_dim)

    def forward(self, x):
        # x is a feature vector of shape (d,)

        h1 = np.matmul(self.W1, x) + self.b1      # "pre-activations" for first hidden layer
        h1 = self.relu(h1)                        # values of first hidden layer

        h2 = np.matmul(self.W2, h1) + self.b2     # "pre-activations" for second hidden layer
        h2 = self.relu(h2)                        # values of second hidden layer

        out = np.matmul(self.beta, h2) + self.b3  # "scores" before softmax of output
        out = self.softmax(out)

        return out
```

Randomly initialize all parameters (weights and biases)

- We won't look at learning the network (backprop) in detail here
- Shapes need to reflect the sizes of hidden layers etc.

```python
class NeuralNetwork:
    def __init__(self, input_dim, h1_dim, h2_dim, n_classes):

        # Weights and biases in first hidden layer
        self.b1 = np.random.randn()
        self.W1 = np.random.randn(h1_dim, input_dim)

        # Weights and biases in second hidden layer
        self.b2 = np.random.randn()
        self.W2 = np.random.randn(h2_dim, h1_dim)

        # Weights and biases in output layer
        self.b3 = np.random.randn()
        self.beta = np.random.randn(n_classes, h2_dim)

    def forward(self, x):
        # x is a feature vector of shape (d,)

        h1 = np.matmul(self.W1, x) + self.b1      # "pre-activations" for first hidden layer
        h1 = self.relu(h1)                        # values of first hidden layer

        h2 = np.matmul(self.W2, h1) + self.b2     # "pre-activations" for second hidden layer
        h2 = self.relu(h2)                        # values of second hidden layer

        out = np.matmul(self.beta, h2) + self.b3  # "scores" before softmax of output
        out = self.softmax(out)

        return out
```

Compute values of first hidden layer

- all hidden notes computed simultaneously via vectorization

$$\mathbf{h}_1 = g(\mathbf{W}_1\mathbf{x} + \mathbf{w}_{10})$$

```python
class NeuralNetwork:
    def __init__(self, input_dim, h1_dim, h2_dim, n_classes):

        # Weights and biases in first hidden layer
        self.b1 = np.random.randn()
        self.W1 = np.random.randn(h1_dim, input_dim)

        # Weights and biases in second hidden layer
        self.b2 = np.random.randn()
        self.W2 = np.random.randn(h2_dim, h1_dim)

        # Weights and biases in output layer
        self.b3 = np.random.randn()
        self.beta = np.random.randn(n_classes, h2_dim)

    def forward(self, x):
        # x is a feature vector of shape (d,)

        h1 = np.matmul(self.W1, x) + self.b1      # "pre-activations" for first hidden layer
        h1 = self.relu(h1)                        # values of first hidden layer

        h2 = np.matmul(self.W2, h1) + self.b2     # "pre-activations" for second hidden layer
        h2 = self.relu(h2)                        # values of second hidden layer

        out = np.matmul(self.beta, h2) + self.b3  # "scores" before softmax of output
        out = self.softmax(out)

        return out
```

Compute values of second hidden layer

- all hidden notes computed simultaneously via vectorization

$$\mathbf{h}_l = g(\mathbf{W}_l \mathbf{h}_{l-1} + \mathbf{w}_{l0})$$

```python
class NeuralNetwork:
    def __init__(self, input_dim, h1_dim, h2_dim, n_classes):

        # Weights and biases in first hidden layer
        self.b1 = np.random.randn()
        self.W1 = np.random.randn(h1_dim, input_dim)

        # Weights and biases in second hidden layer
        self.b2 = np.random.randn()
        self.W2 = np.random.randn(h2_dim, h1_dim)

        # Weights and biases in output layer
        self.b3 = np.random.randn()
        self.beta = np.random.randn(n_classes, h2_dim)

    def forward(self, x):
        # x is a feature vector of shape (d,)

        h1 = np.matmul(self.W1, x) + self.b1      # "pre-activations" for first hidden layer
        h1 = self.relu(h1)                        # values of first hidden layer

        h2 = np.matmul(self.W2, h1) + self.b2     # "pre-activations" for second hidden layer
        h2 = self.relu(h2)                        # values of second hidden layer

        out = np.matmul(self.beta, h2) + self.b3  # "scores" before softmax of output
        out = self.softmax(out)

        return out
```

Compute outputs of network; one probability per possible class

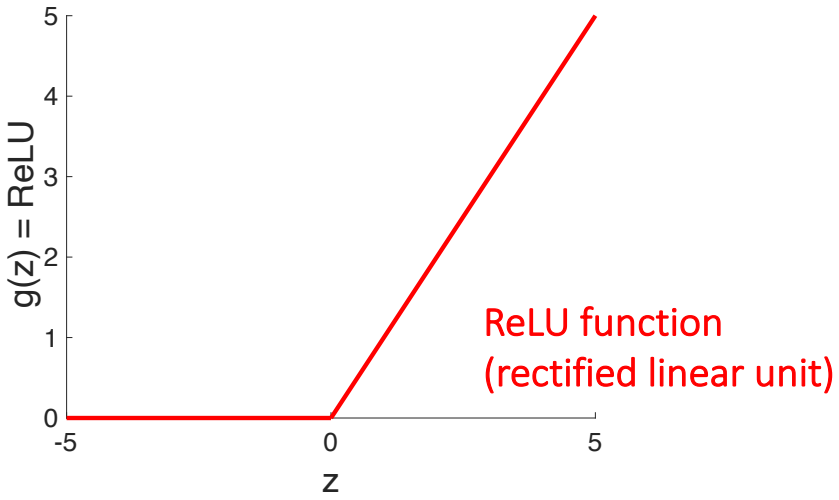
$$f_k(\mathbf{x};\boldsymbol{\theta}) = \text{softmax}\left(\boldsymbol{\beta}_k\mathbf{h} + \beta_{0k}\right)$$

```
def relu(self, z):
    # Activation function -- this is the ReLU function
    return np.minimum(z, 0)

def softmax(self, z):
    # Softmax function -- turns unnormalized scores into a probability vector
    exp_z = np.exp(z)
    return exp_z / np.sum(exp_z)
```

The ReLU function is very simple:

$z > 0 : g(z) = z$

$z <= 0 : g(z) = 0$



Our numpy implementation computes the ReLU function on a vector

- i.e. applies the ReLU to each entry in the vector z simultaneously

```python
def relu(self, z):
    # Activation function -- this is the ReLU function
    return np.minimum(z, 0)

def softmax(self, z):
    # Softmax function -- turns unnormalized scores into a probability vector
    exp_z = np.exp(z)
    return exp_z / np.sum(exp_z)
```

$$\text{softmax}(z_k) = \frac{e^{z_k}}{\sum_{m=1}^{K} e^{z_m}}$$

Turns the "scores" coming from the final hidden layer into a probability distribution over classes

```python
out = np.matmul(self.beta, h2) + self.b3  # "scores" before softmax of output
out = self.softmax(out)

return out
```

```
def relu(self, z):
    # Activation function -- this is the ReLU function
    return np.minimum(z, 0)

def softmax(self, z):
    # Softmax function -- turns unnormalized scores into a probability vector
    exp_z = np.exp(z)
    return exp_z / np.sum(exp_z)
```

$$\text{softmax}(z_k) = \frac{e^{z_k}}{\sum_{m=1}^{K} e^{z_m}}$$

Turns the "scores" coming from the final hidden layer into a probability distribution over classes

Our implementation is vectorized: computes the entire softmax vector at once

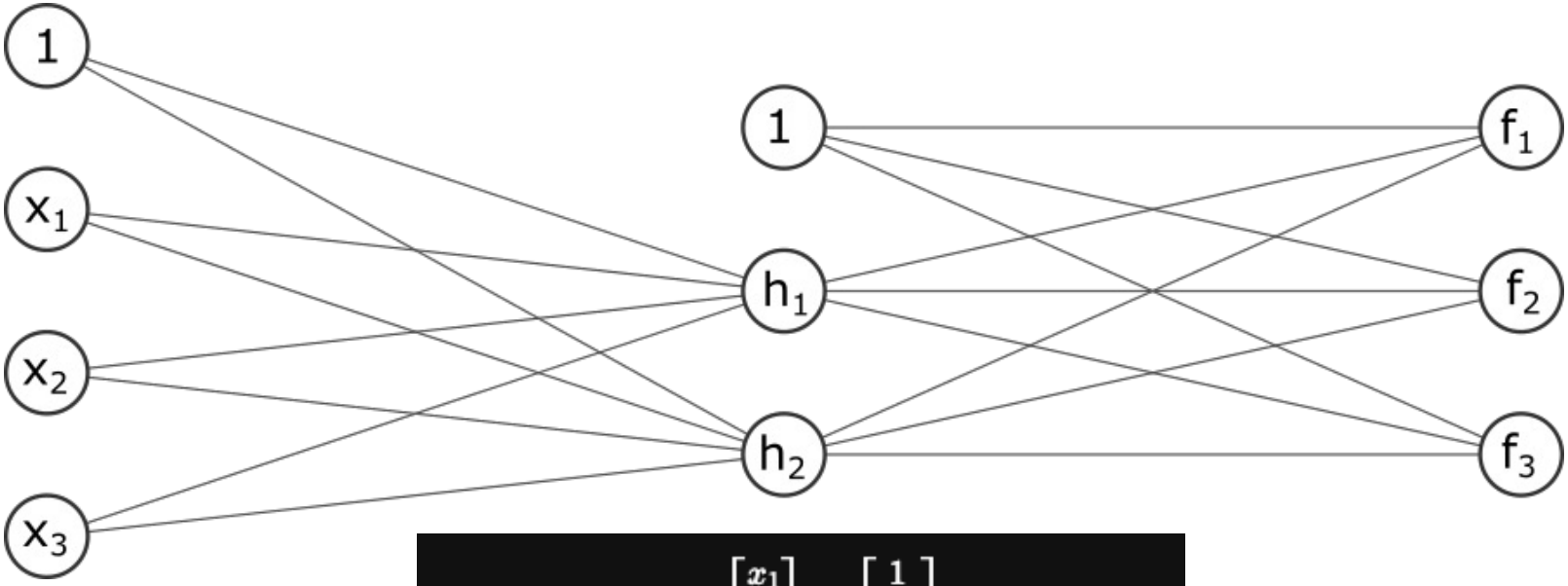
```python
class NeuralNetwork:
    def __init__(self, input_dim, h1_dim, h2_dim, n_classes):

        # Weights and biases in first hidden layer
        self.b1 = np.random.randn()
        self.W1 = np.random.randn(h1_dim, input_dim)

        # Weights and biases in second hidden layer
        self.b2 = np.random.randn()
        self.W2 = np.random.randn(h2_dim, h1_dim)

        # Weights and biases in output layer
        self.b3 = np.random.randn()
        self.beta = np.random.randn(n_classes, h2_dim)

    def forward(self, x):
        # x is a feature vector of shape (d,)

        h1 = np.matmul(self.W1, x) + self.b1      # "pre-activations" for first hidden layer
        h1 = self.relu(h1)                        # values of first hidden layer

        h2 = np.matmul(self.W2, h1) + self.b2     # "pre-activations" for second hidden layer
        h2 = self.relu(h2)                        # values of second hidden layer

        out = np.matmul(self.beta, h2) + self.b3  # "scores" before softmax of output
        out = self.softmax(out)

        return out
```

Equivalent network in sklearn:

```python
from sklearn.neural_network import MLPClassifier
nn = MLPClassifier(hidden_layer_sizes=(h1_dim, h2_dim), activation='relu')
```

HW3 problem 1:

- compute the “forward pass” on a very simple network by hand



$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \\ -2 \end{bmatrix}$$

$$W = \begin{bmatrix} w_{01} & w_{11} & w_{21} & w_{31} \\ w_{02} & w_{12} & w_{22} & w_{32} \end{bmatrix} = \begin{bmatrix} 1 & -1 & 0 & 6 \\ 2 & 1 & 1 & 3 \end{bmatrix}$$

$$B = \begin{bmatrix} \beta_{01} & \beta_{11} & \beta_{21} \\ \beta_{02} & \beta_{12} & \beta_{22} \\ \beta_{03} & \beta_{13} & \beta_{23} \end{bmatrix} = \begin{bmatrix} 6 & -1 & 0 \\ 5 & 0 & 2 \\ 2 & 1 & 1 \end{bmatrix}$$

Questions?

Neural Network Recap

Neural Networks in Code

Tuning a Network

Hyperparameters

- Neural networks have very many hyperparameters:
  - Regularization weight α
    - And type of regularization: L1, L2, etc.
  - Hyperparameters of optimizer
    - Learning rate
    - Batch size b for stochastic gradient descent
  - Network architecture for neural network models
    - Number of layers, size of layers, type of non-linearity, etc.
    - More layers, bigger layers → more complex models

Complexity and Accuracy Tradeoffs

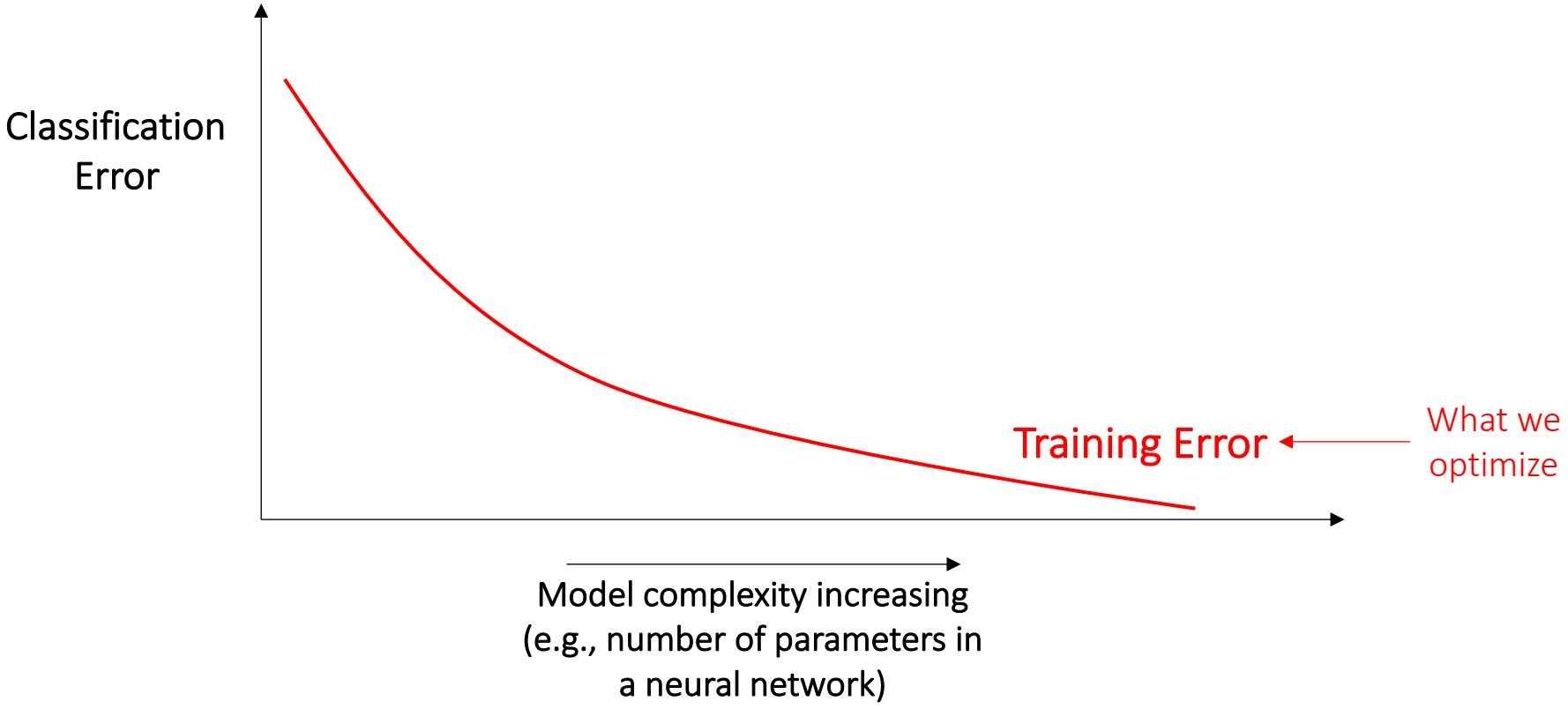
There is a fundamental trade-off between model complexity and model accuracy in machine learning models

As models get more complex, they can fit the training data perfectly ….but their performance on new test data may get worse

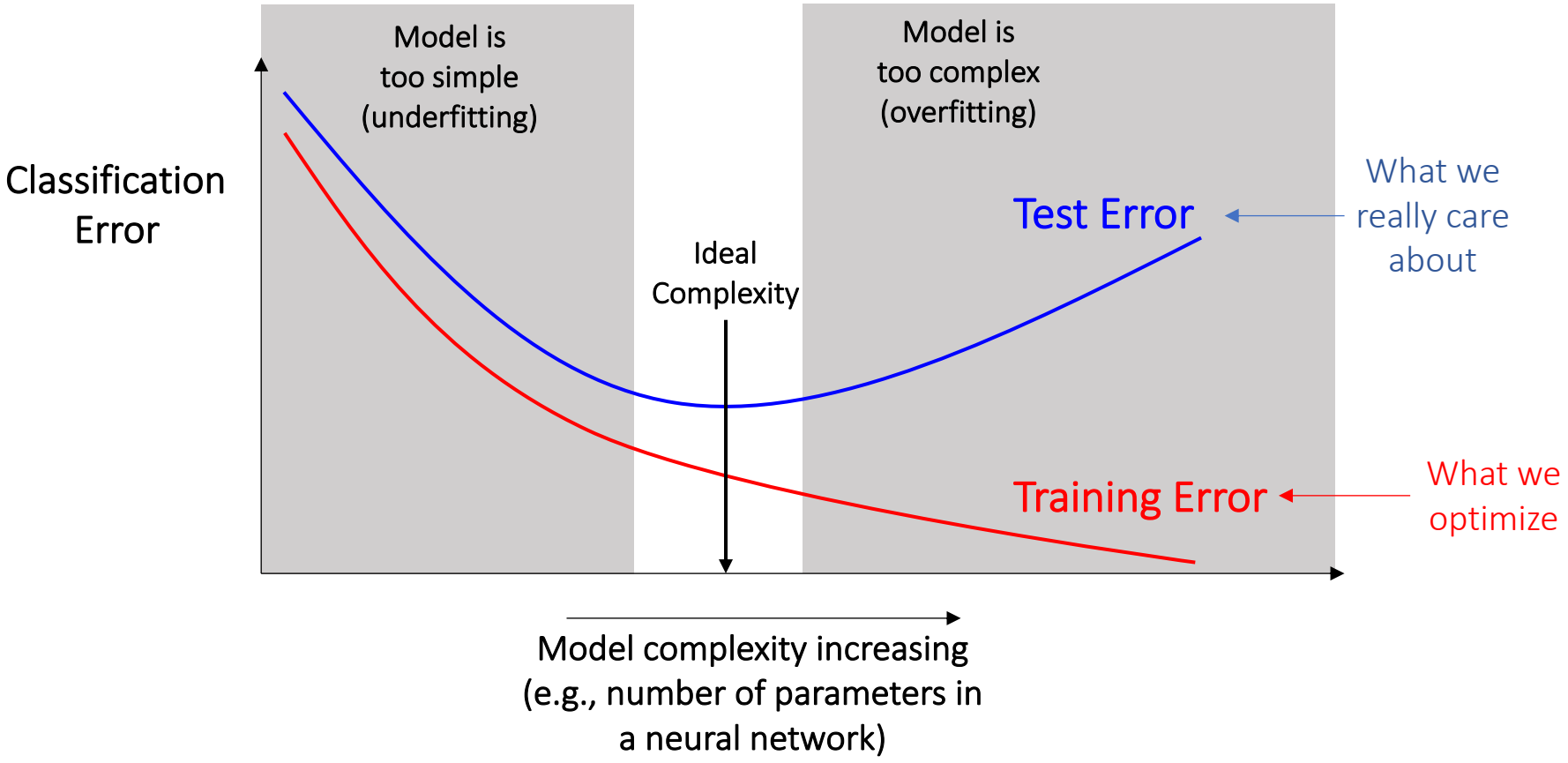
For neural networks, increasing the size of the network (the number of parameters) increases the complexity of the model

(but the basic complexity/performance tradeoffs exists for any type of model where we can vary the complexity of the model)

Complexity and Accuracy Tradeoffs



Complexity and Accuracy Tradeoffs



Theoretical Aspects: Bias and Variance

Bias of a model = bias in terms of approximating true boundary

Variance of a model = sensitivity of model training to different data sets

Classic result in machine learning:

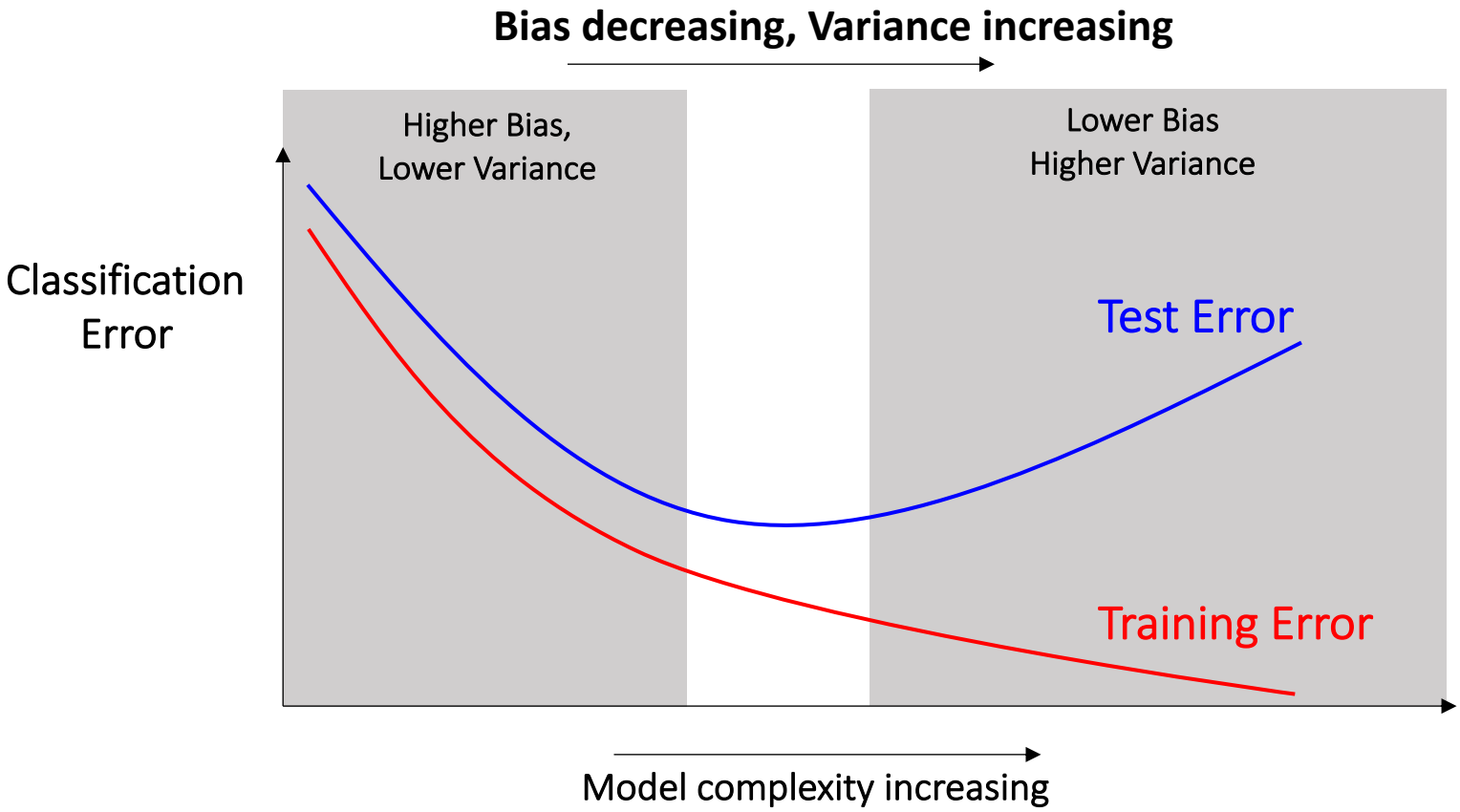
Test Error is a function of Bias + Variance

Bias-Variance Tradeoff:

Simpler models => higher bias, lower variance

Complex models => lower bias, higher variance

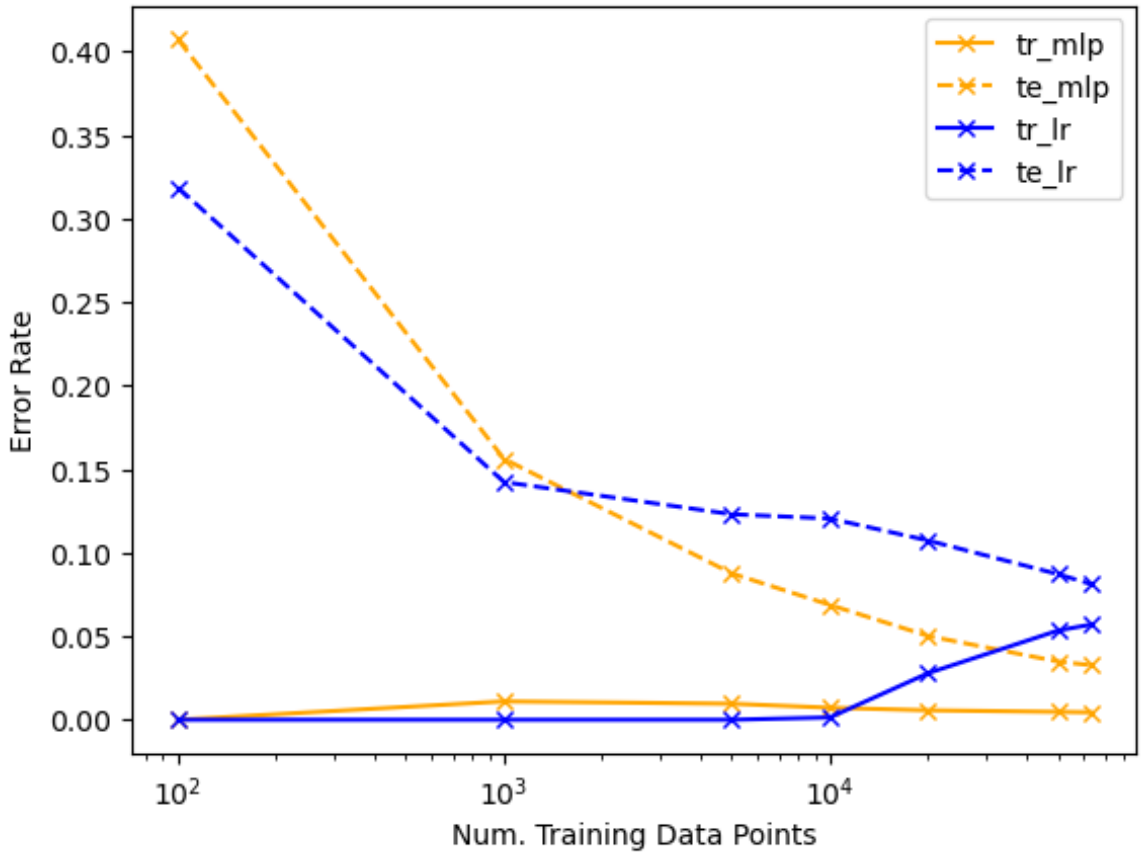
Bias-Variance Tradeoff



Bias-Variance Tradeoff

HW3 Problem 2.2: Reproduce this figure on MNIST

- High bias, low variance: Logistic Regression
- Low bias, high variance: Neural Networks



Advice on Model Complexity

1. More complex models can always fit better to training data:
   => use validation data to fairly compare models

2. Always include a relatively simple model (e.g., logistic) in your search

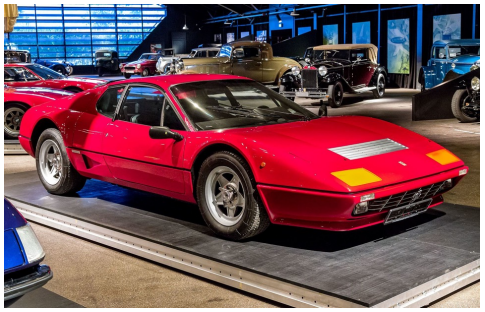
3. Larger datasets allow for more complex models
   …. conversely, for smaller datasets simpler models are often best

4. Be careful with small datasets: easy to overfit

5. Regularization can help avoid overfitting (can be used with any model, any loss function, e.g., with neural networks and log-loss)

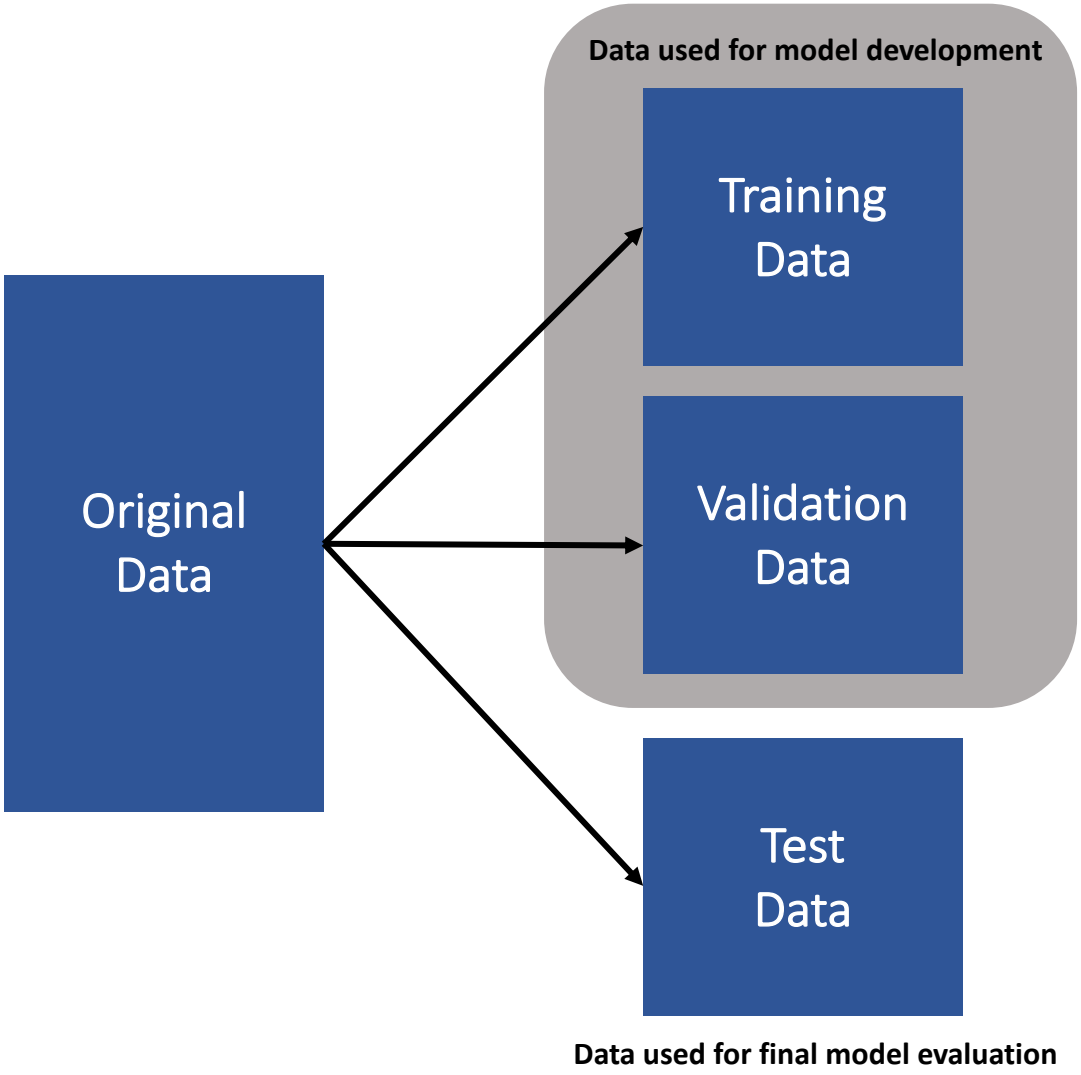
Logistic/linear model

"Shallow" neural networks

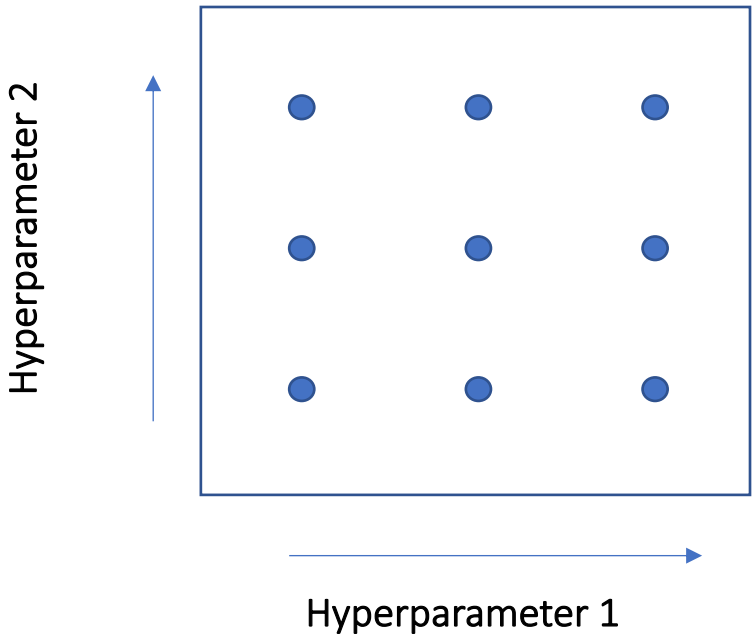


Large deep neural networks

Recall: Train, Validation, Test Datasets



Grid Search

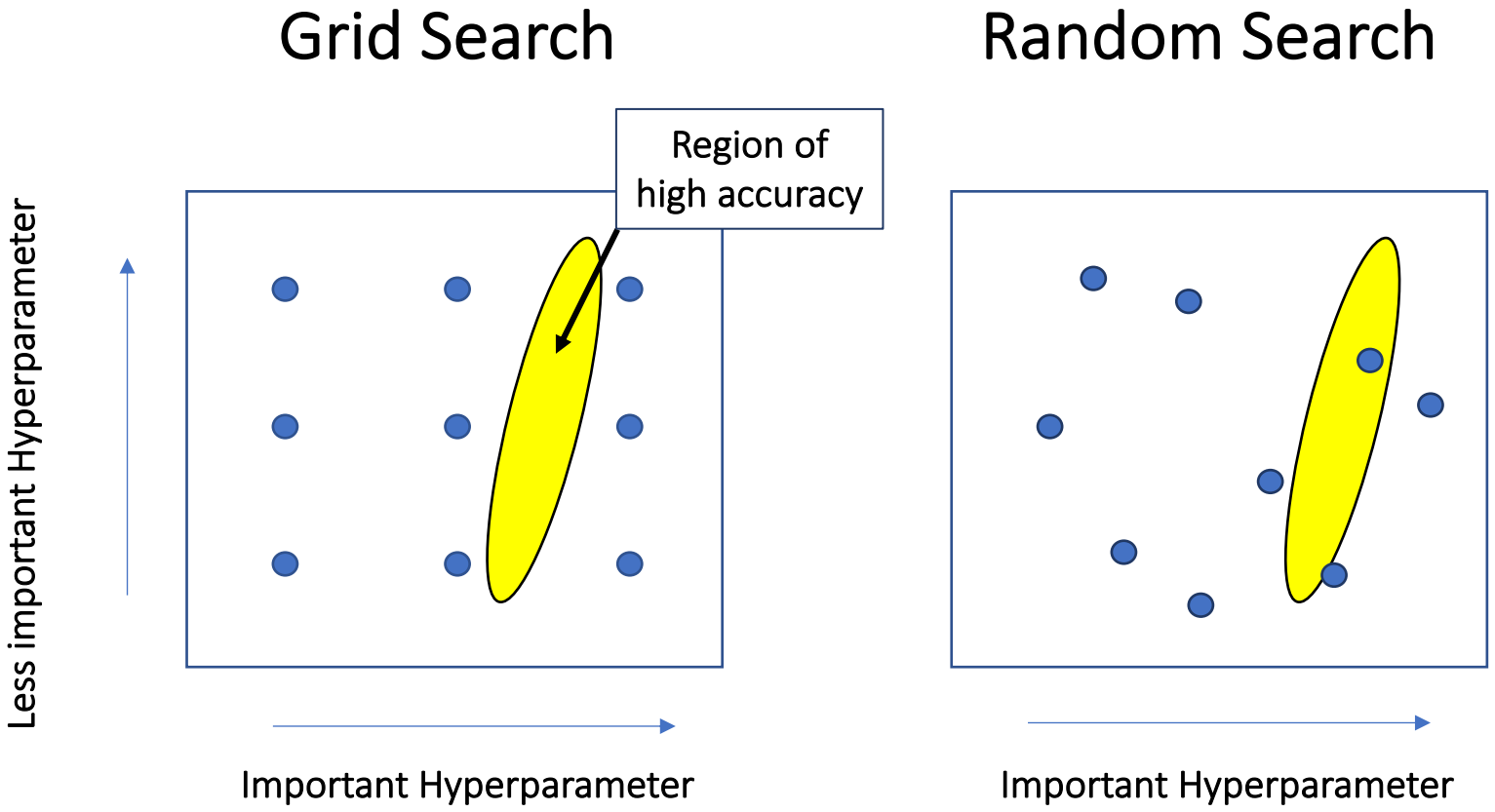


Grid Search Procedure:

- For real-valued hyperparameters
  - Set [min, max] range for each hyperparameter
  - Set number of values within each range
- For categorical hyperparameters, determine values
- For every possible combination ("grid value")
  - Fit the model on the training data
  - Compute accuracy on validation data
- Select the hyperparameter values that give highest classification accuracy **on validation data**

Computationally intensive:

e.g., D hyperparameters, each taking M values
=> M^D different models to train



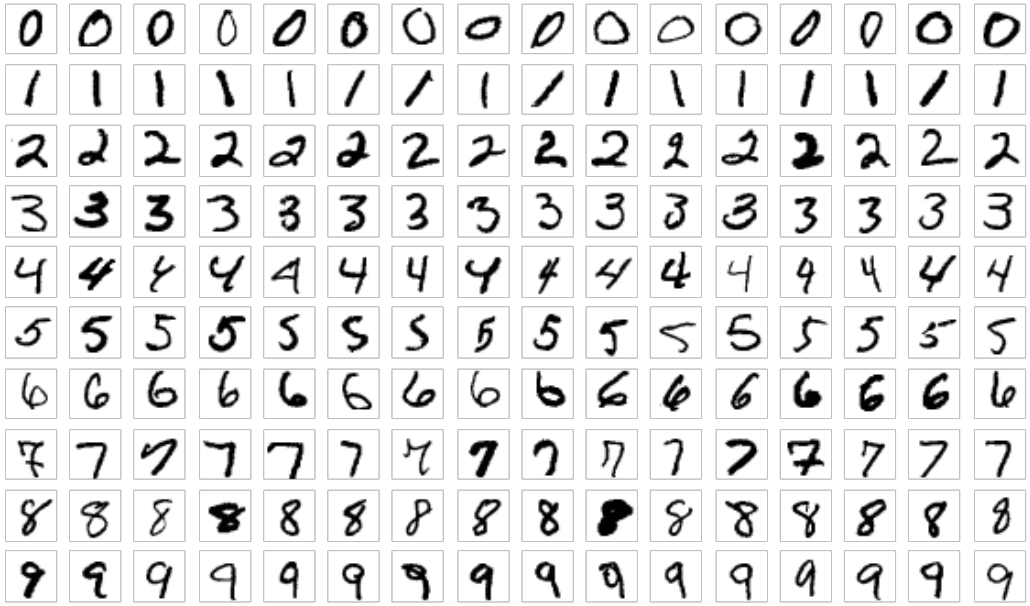
For random search, can select hyperparameter values by sampling from a uniform distribution over range of values

Questions?

HW3

Problem 2.4: Tuning a Neural Network

- Your job: achieve a validation error rate of <5% on the MNIST dataset with a neural network
- Not easy! Can receive most of the credit for training at least 10 hyperparameter settings



HW3

Problem 2.4: Tuning a Neural Network

sklearn.neural_network.MLPClassifier

class `sklearn.neural_network.`**`MLPClassifier`**(*hidden_layer_sizes=(100,), activation='relu', *, solver='adam', alpha=0.0001, batch_size='auto', learning_rate='constant', learning_rate_init=0.001, power_t=0.5, max_iter=200, shuffle=True, random_state=None, tol=0.0001, verbose=False, warm_start=False, momentum=0.9, nesterovs_momentum=True, early_stopping=False, validation_fraction=0.1, beta_1=0.9, beta_2=0.999, epsilon=1e-08, n_iter_no_change=10, max_fun=15000*) [source]

Many different hyperparameters you can play with

- Activation function?
- Number of hidden layers?
- Size of hidden layers?
- Learning rate?
- Regularization function and strength?
- ….

HW3

Problem 2.4: Tuning a Neural Network

sklearn.neural_network.MLPClassifier

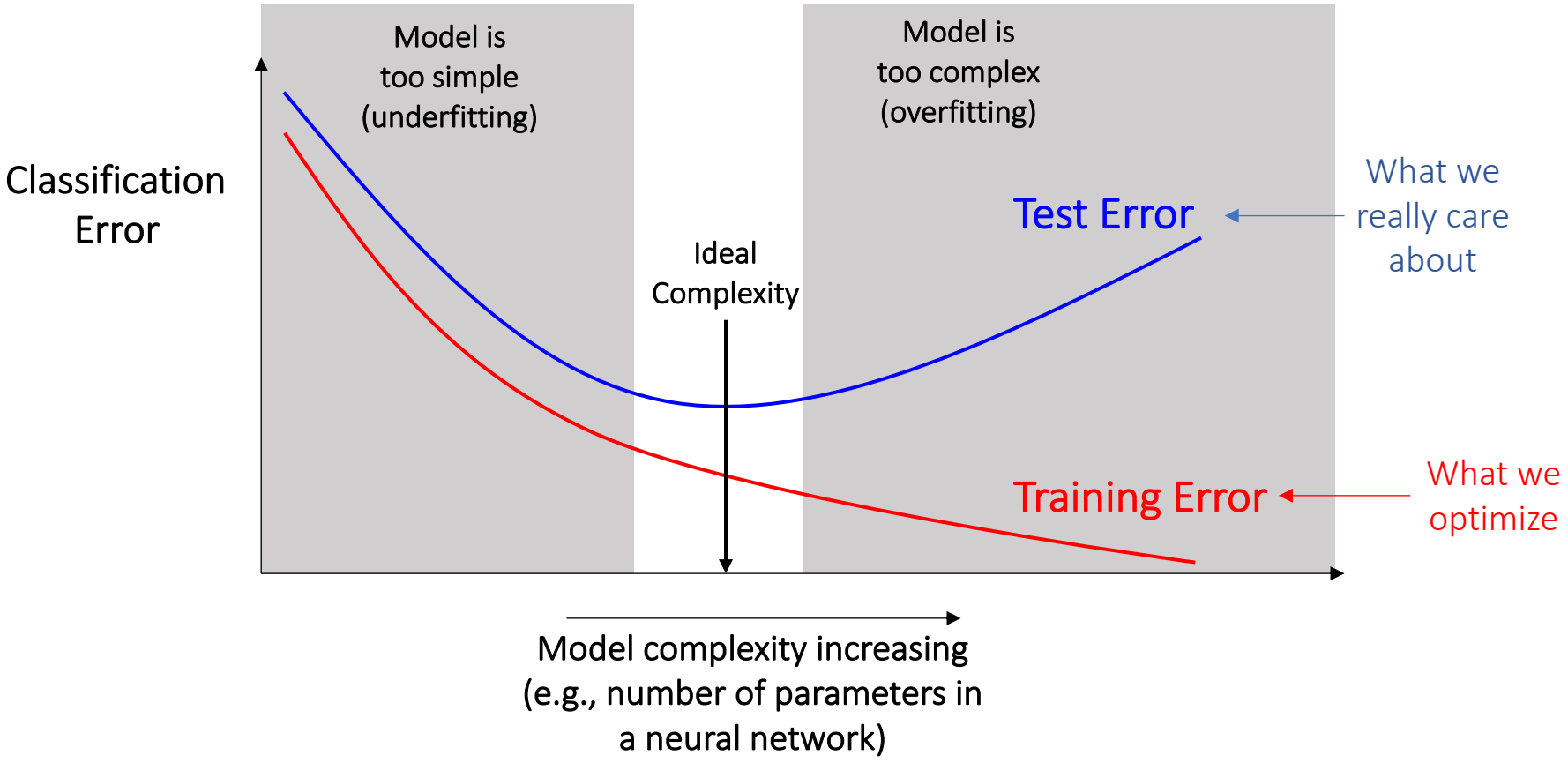
class `sklearn.neural_network.`**`MLPClassifier`**(*hidden_layer_sizes=(100,), activation='relu', *, solver='adam', alpha=0.0001, batch_size='auto', learning_rate='constant', learning_rate_init=0.001, power_t=0.5, max_iter=200, shuffle=True, random_state=None, tol=0.0001, verbose=False, warm_start=False, momentum=0.9, nesterovs_momentum=True, early_stopping=False, validation_fraction=0.1, beta_1=0.9, beta_2=0.999, epsilon=1e-08, n_iter_no_change=10, max_fun=15000*) [source]

Many different hyperparameters you can play with

Training neural networks is *slow*

- Exhaustive grid search not feasible
- Use the notions of *over/under* fitting to guide your search

Complexity and Accuracy Tradeoffs



Overfitting and Underfitting

How can we determine if a model is overfitting or underfitting?

- A heuristic (but not set-in-stone) rule:

	High Training Error	Low Training Error
High Val. Error	**Underfitting**	**Overfitting**
Low Val. Error	Bug in your code ☺	Appropriate Fit

You can use this to iteratively assess model performance, and modify hyperparameters accordingly

- Example: try some initial guess for hyperparameters
- See your model is underfitting? Make it more complex!
- See your model is overfitting? Reduce complexity or regularize!

Questions?

Wrapup

- End of our discussion on feed-forward neural networks
  - Complex, large models
  - Most useful on hard problems with large datasets
    - But not appropriate for all classification problems!
  - Extension of logistic classifiers

- Hyperparameter selection is difficult
  - Think about *over/under* fitting
  - Can try random search

- Next lecture
  - Convolutional neural networks
  - A new (to us) architecture designed for image data