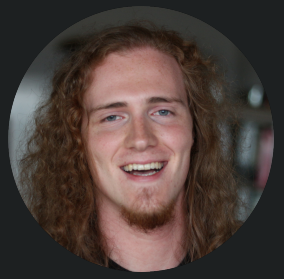


LUKASSCHMELZEISEN.

lukas@uni-koblenz.de
lschmelzeisen.com

24 Sep 2018

Introduction to Supervised Learning with TensorFlow

Find the most up-to-date version of this talk and all code at:
https://github.com/lschmelzeisen/talk-supervised-learning-tensorflow

License

Small Warning

- This talk will contain math and code!
  - Understanding the math will be easy, basic linear algebra is sufficient
  - Understanding the code is no requirement to follow the talk, but not hard because we go through it line by line
- If I go to fast, slow me down!
- If you have questions, ask immediately!

Code

- All code in these slides can be found at
  https://github.com/lschmelzeisen/talk-supervised-learning-tensorflow
- If you want to try out the code without having to worry about installing dependencies, you can use **Google Colaboratory**
  - Free jupyter notebook environment for research and education
- Links that directly open the respective code files in Colaboratory are provided in the README on GitHub

Math Notation

Should you forget a variable's type, you can infer it from it's name!

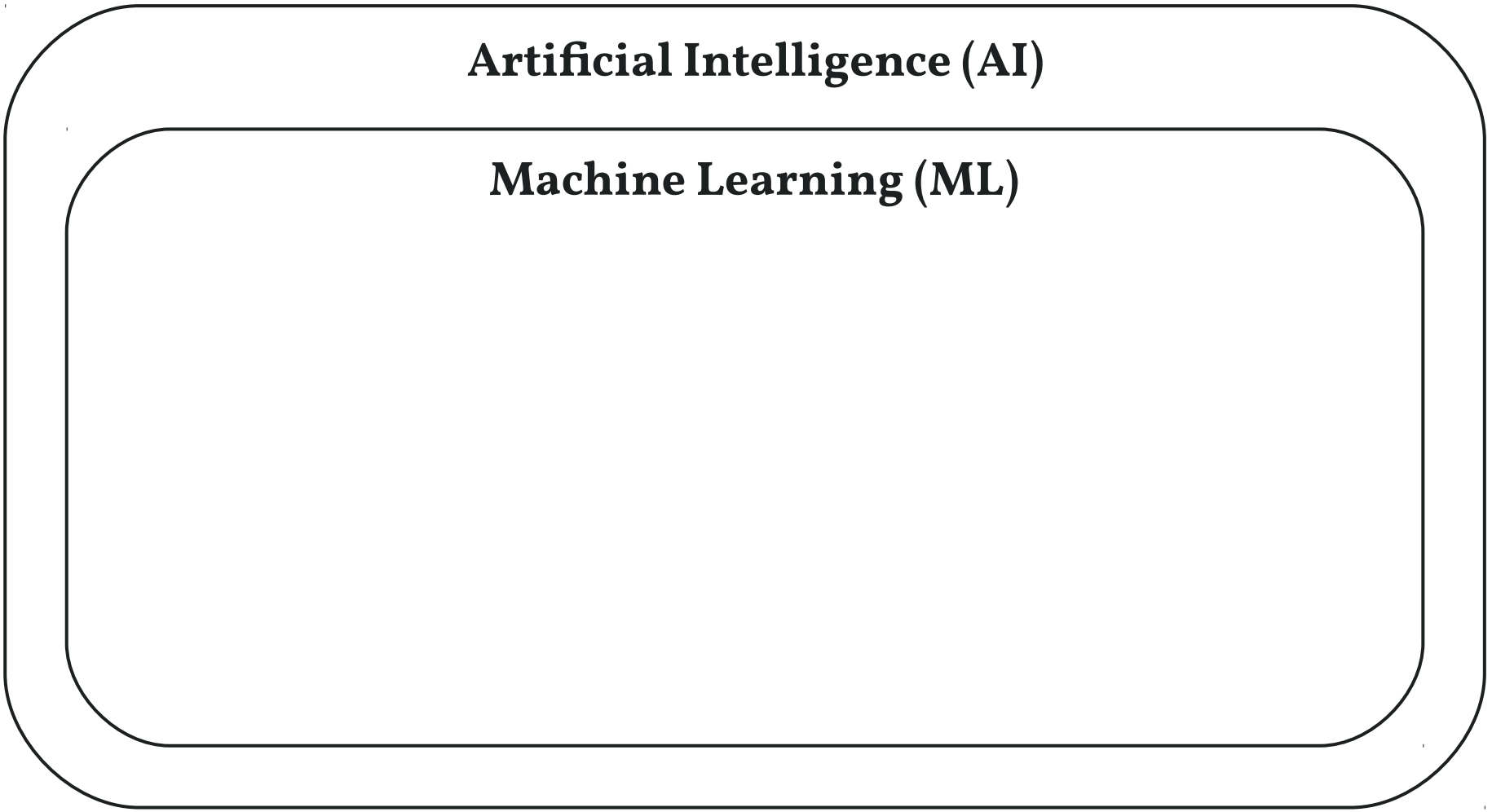
- Lowercase italic letters are **scalars**: $x = 3$
- Lowercase bold letters are (row-) **vectors**: $\mathbf{v} = \begin{bmatrix} 4 & -2 & 5 \end{bmatrix}$
- Uppercase bold letters are **matrices**: $\mathbf{M} = \begin{bmatrix} 1 & 7 \\ 3 & 8 \end{bmatrix}$

What is this talk about?

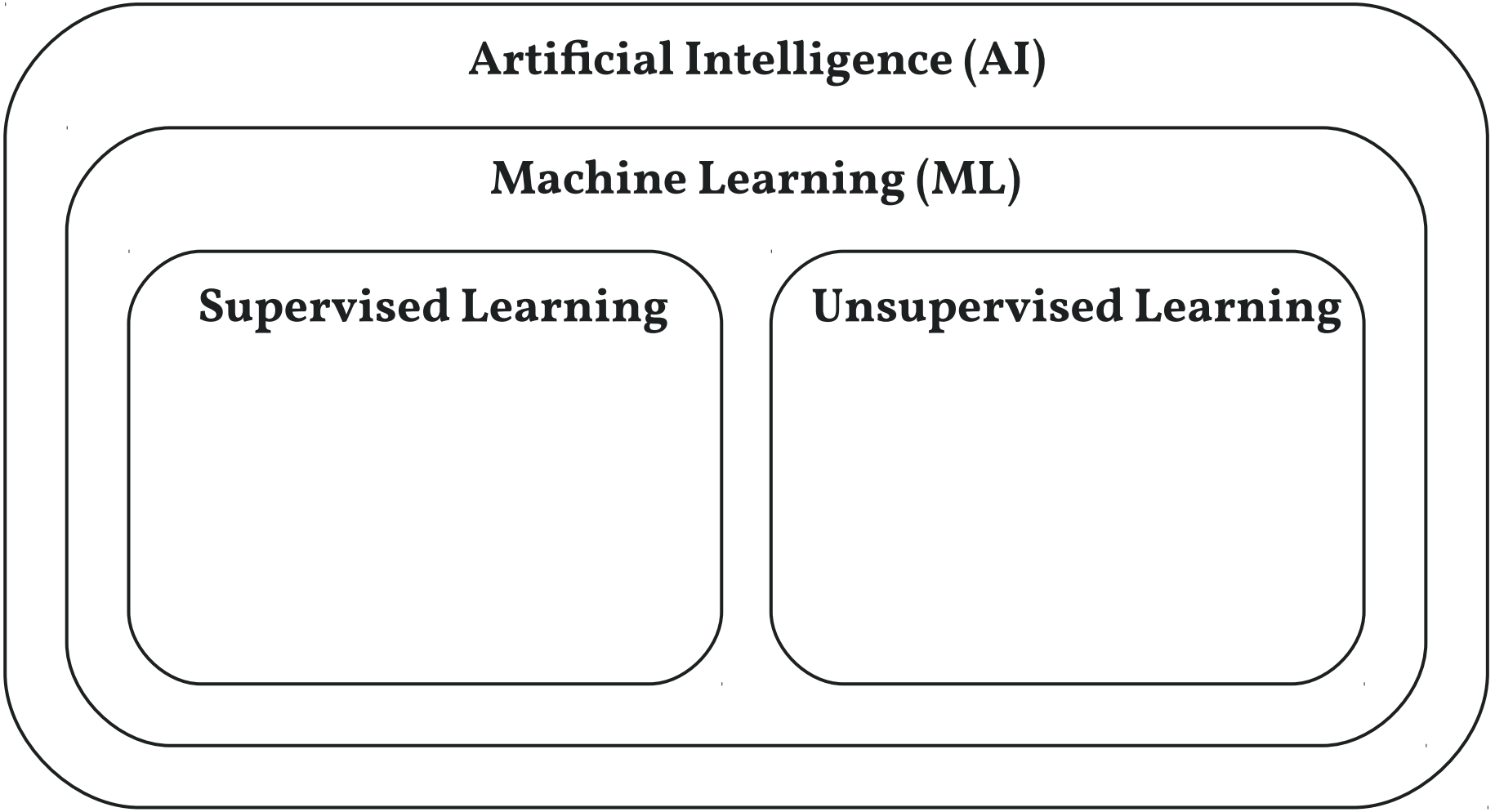
What is this talk about?

Artificial Intelligence (AI)

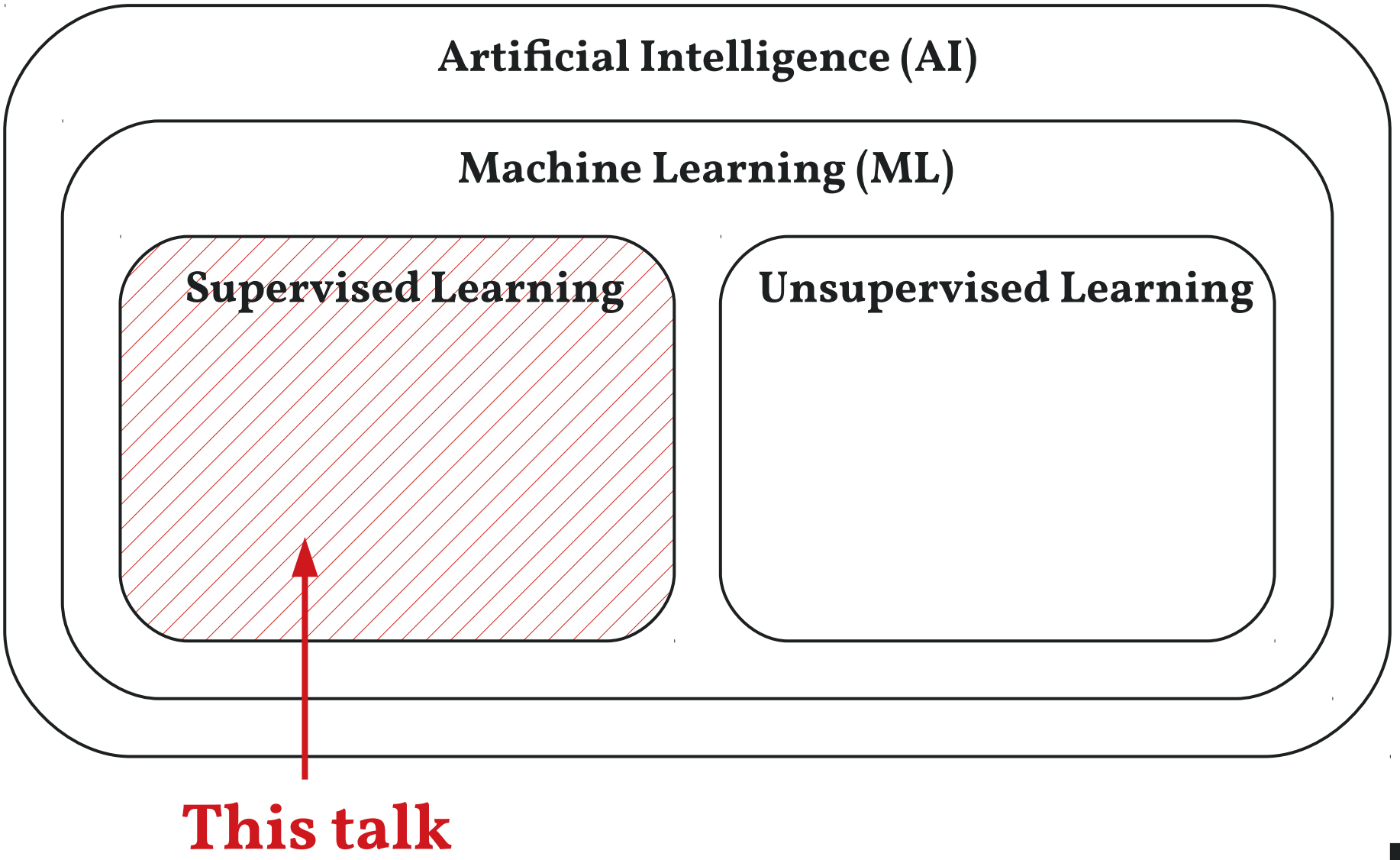
What is this talk about?



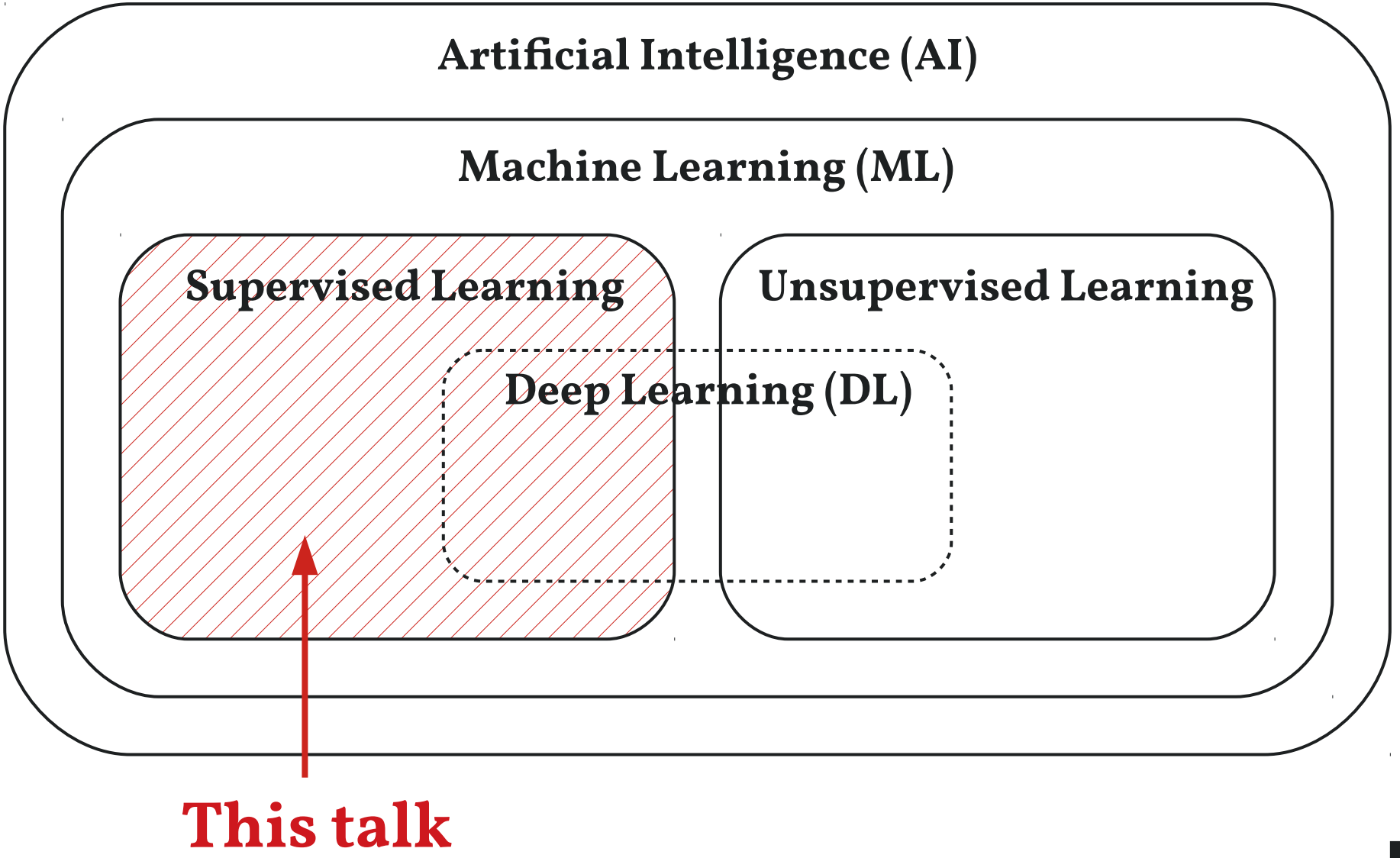
What is this talk about?



What is this talk about?



What is this talk about?



Machine Learning

- Often, finding an algorithm to solve a certain problem can be very difficult (e.g. image classification or machine translation)
- However, sometimes it is comparatively easy to collect input/target-examples for these problems

Machine Learning

- Often, finding an algorithm to solve a certain problem can be very difficult (e.g. image classification or machine translation)
- However, sometimes it is comparatively easy to collect input/target-examples for these problems
  - **Airplane:**
  - **Automobile:**
  - **Bird:**
  - **Cat:**
  - **Deer:**
  - **Dog:**
  - **Frog:**
  - **Horse:**
  - **Ship:**
  - **Truck:**

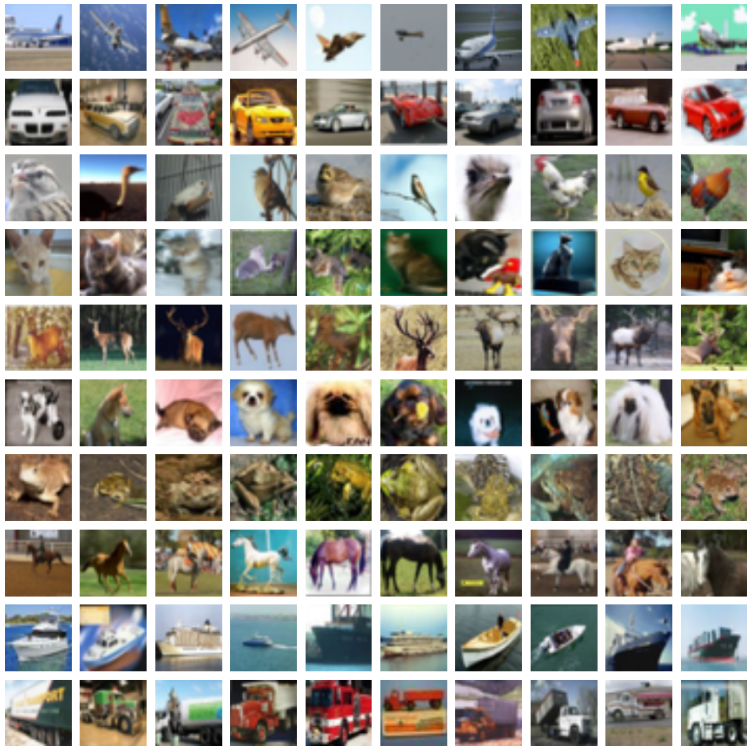
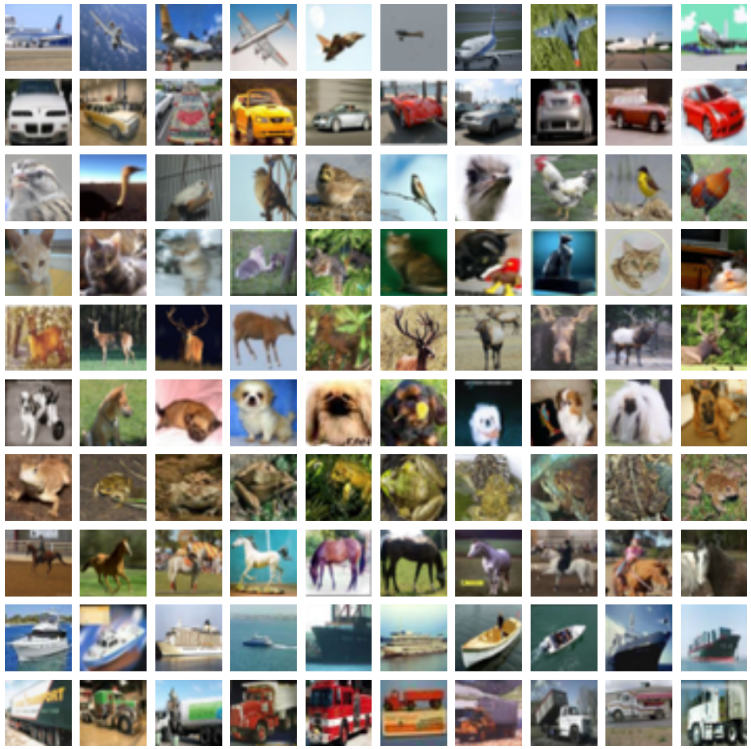


Image of CIFAR-10 dataset, from: https://www.cs.toronto.edu/~kriz/cifar.html

Machine Learning

- Often, finding an algorithm to solve a certain problem can be very difficult (e.g. image classification or machine translation)
- However, sometimes it is comparatively easy to collect input/target-examples for these problems
  - **Airplane:**
  - **Automobile:**
  - **Bird:**
  - **Cat:**
  - **Deer:**
  - **Dog:**
  - **Frog:**
  - **Horse:**
  - **Ship:**
  - **Truck:**



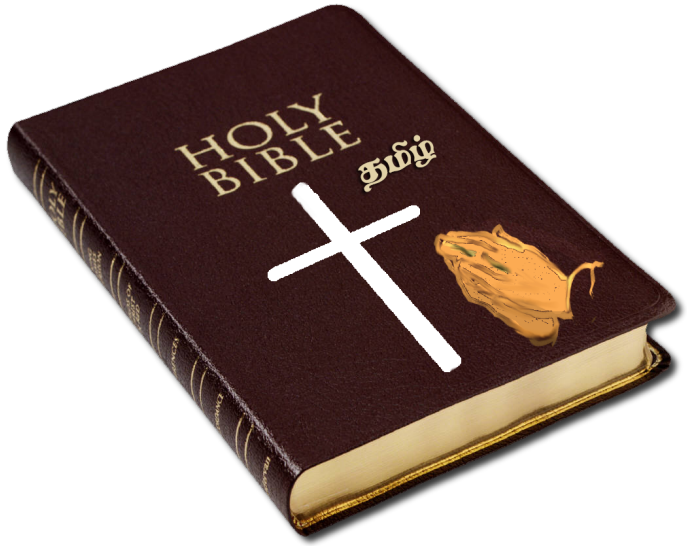


Image of CIFAR-10 dataset, from: https://www.cs.toronto.edu/~kriz/cifar.html
Image of bible, from: https://purepng.com/photo/19065/objects-holy-bible (CC0)

Machine Learning

- Often, finding an algorithm to solve a certain problem can be very difficult (e.g. image classification or machine translation)
- However, sometimes it is comparatively easy to collect input/target-examples for these problems
- Machine Learning: try to automatically find a function that produces the intended target given the input (approximately)

Supervised Learning

- **Dataset** of input-target-examples:

  $$D = \{(\mathbf{x}_1, \mathbf{y}_1), (\mathbf{x}_2, \mathbf{y}_2), \ldots, (\mathbf{x}_N, \mathbf{y}_N)\}$$

- **Model function** that approximates the target given an input $\mathbf{x}_i$ and parameters Θ: $\hat{\mathbf{y}}_i = f(\mathbf{x}_i; \Theta)$

- **Loss function** that measures distance of expected target $\mathbf{y}_i$ to predicted output $\hat{\mathbf{y}}_i$: $L(\mathbf{y}_i, \hat{\mathbf{y}}_i)$

- **Training algorithm** that finds parameters minizing the loss:

  $$\Theta_{\text{optimal}} = \arg\min_{\Theta} \sum_{(\mathbf{x}_i, \mathbf{y}_i) \in D} L(\mathbf{y}_i, f(\mathbf{x}_i; \Theta))$$

Supervised Learning

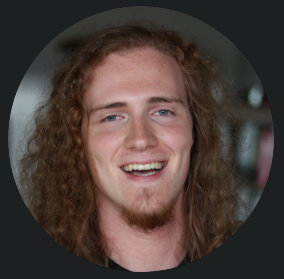
- Shape of data $(\mathbf{x}_i, \mathbf{y}_i) \in D$ is important!
- For inputs: usually $\mathbf{x}_i = \begin{bmatrix} x_{i1} & x_{i2} & \dots & x_{id_{\text{in}}} \end{bmatrix} \in \mathbb{R}^{d_{\text{in}}}$
  - Each entry x_{ij} is called a **feature**
  - Each $\mathbf{x}_i$ is called a **feature vector**
- For targets:
  - If $\mathbf{y}_i \in \mathbb{R}$, we speak of a **regression problem**
  - If the set of all $\mathbf{y}_i$ is descrete, we instead speak of a **classification problem**
    - The different $\mathbf{y}_i$ are called **classes**

In Contrast: Unsupervised Learning

- Same structure as supervised learning, but without targets
- Need to find structure in unlabeled data
- Usually via clustering algorithms
- Not part of this talk

Outline

1. Simple Linear Regression
   - » Stochastic Gradient Descent
   - » Computation Graph Abstraction
2. Linear Regression
   - » Tensors
   - » Standardization
   - » Mini-Batch Gradient Descent
3. Logistic Regression
   - » Sigmoid
   - » Softmax
4. Feedforward Neural Network
   - » Dense Layers
   - » Regularization
5. Conclusion



LUKASSCHMELZEISEN.

lukas@uni-koblenz.de
lschmelzeisen.com

24 Sep 2018

Introduction to Supervised Learning with TensorFlow

» Simple Linear Regression

Simple Linear Regression

- Applicable if input and output are scalar:
  - $\mathbf{x}_i = x_i \in \mathbb{R}$
  - $\mathbf{y}_i = y_i \in \mathbb{R}$
- Just two scalar parameters $\Theta = \{m, b\}$
  - slope m
  - y-intercept b
- Model function: $\hat{y}_i = f(x_i; \Theta) = x_i \cdot m + b$
- Loss is usually **squared error**: $L(y_i, \hat{y}_i) = (y_i - \hat{y}_i)^2$

Stochastic Gradient Descent (SGD)

- Optimization algorithm to find parameters Θ that minimize loss function L
- Closed-form solutions for (simple) linear regression exist
- But, stochastic gradient descent will be applicable to all models in this talk
- Guaranteed to converge for convex functions, but can also be applied to non-convex loss functions like neural networks
- Requirement: need to be able to differentiate loss function with respect to parameters $\frac{\partial L(\mathbf{y}_i, \hat{\mathbf{y}}_i)}{\partial \Theta}$

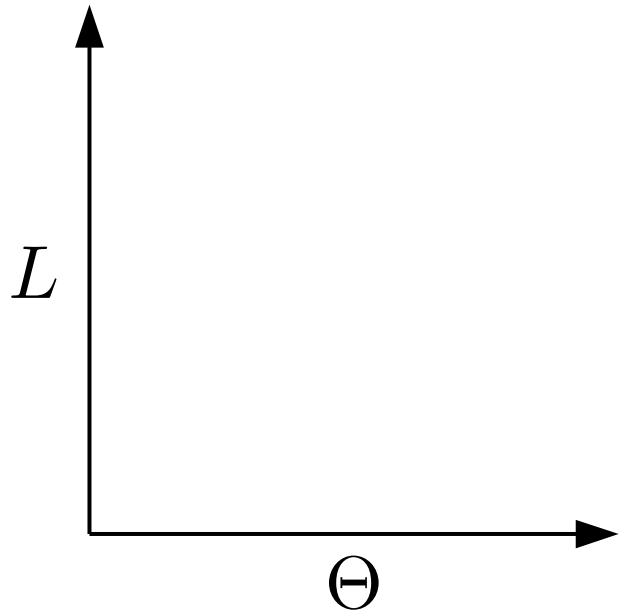
Stochastic Gradient Descent (SGD)

- Optimization algorithm to find parameters Θ that minimize loss function L
- Closed-form solutions for (simple) linear regression exist
- But, stochastic gradient descent will be applicable to all models in this talk
- Guaranteed to converge for convex functions, but can also be applied to non-convex loss functions like neural networks
- Requirement: need to be able to differentiate loss function with respect to parameters $\frac{\partial L(\mathbf{y}_i, \hat{\mathbf{y}}_i)}{\partial \Theta}$

slope of loss function =
direction in which loss decreases
for current parameters

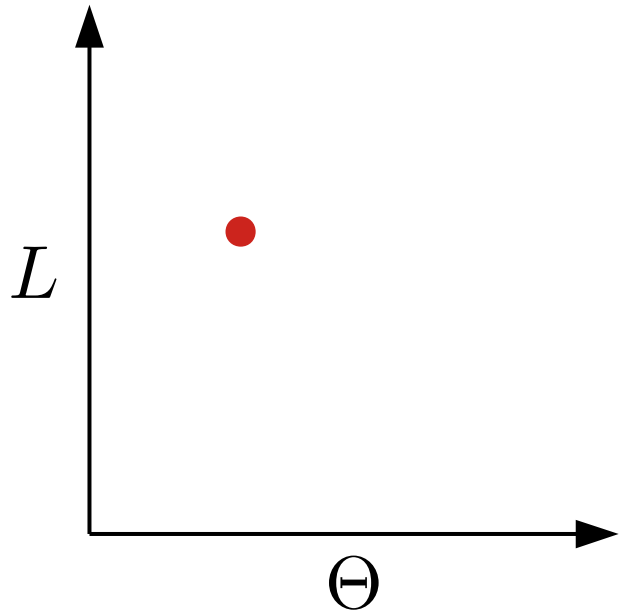
Stochastic Gradient Descent (SGD)

- Setting: unknown relation between loss L and parameters Θ
- Task: find Θ that minimize L given gradient $\frac{\partial L(\mathbf{y}_i, \hat{\mathbf{y}}_i)}{\partial \Theta}$



Stochastic Gradient Descent (SGD)

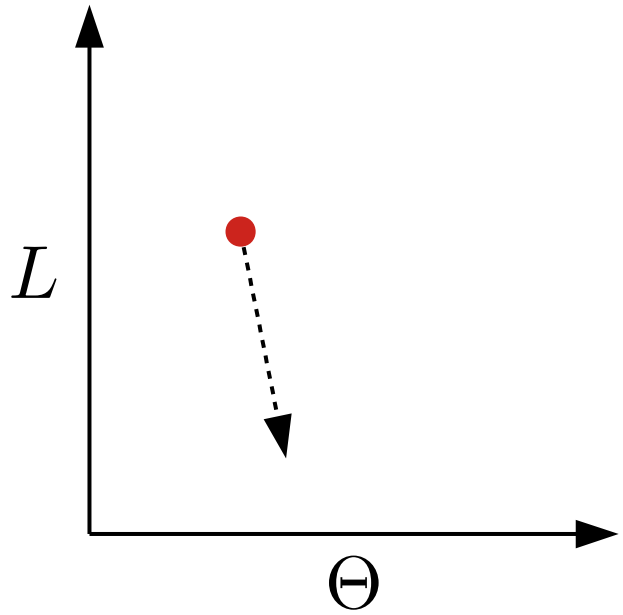
- Setting: unknown relation between loss L and parameters Θ
- Task: find Θ that minimize L given gradient $\frac{\partial L(\mathbf{y}_i, \hat{\mathbf{y}}_i)}{\partial \Theta}$



1. Initialize Θ randomly

Stochastic Gradient Descent (SGD)

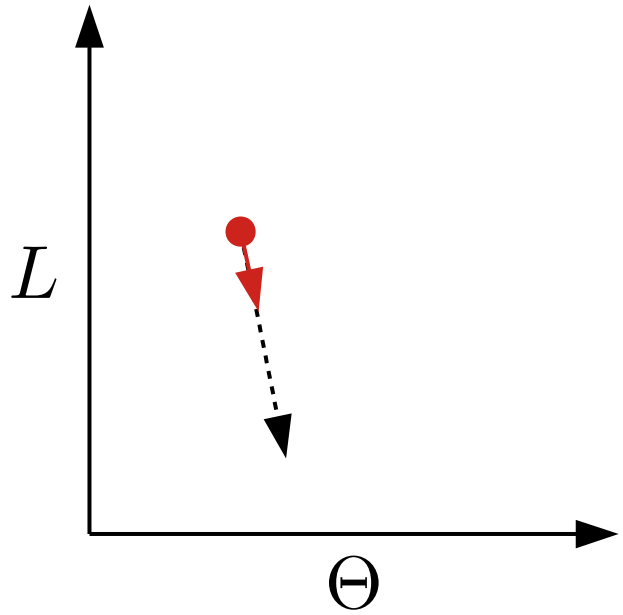
- Setting: unknown relation between loss L and parameters Θ
- Task: find Θ that minimize L given gradient $\frac{\partial L(\mathbf{y}_i, \hat{\mathbf{y}}_i)}{\partial \Theta}$



1. Initialize Θ randomly
2. Compute gradient

Stochastic Gradient Descent (SGD)

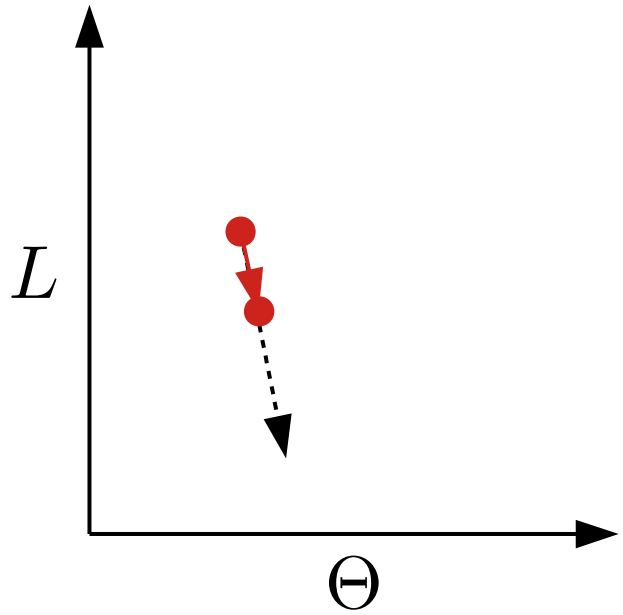
- Setting: unknown relation between loss L and parameters Θ
- Task: find Θ that minimize L given gradient $\frac{\partial L(\mathbf{y}_i, \hat{\mathbf{y}}_i)}{\partial \Theta}$



1. Initialize Θ randomly
2. Compute gradient
3. Move Θ a small step in direction of gradient

Stochastic Gradient Descent (SGD)

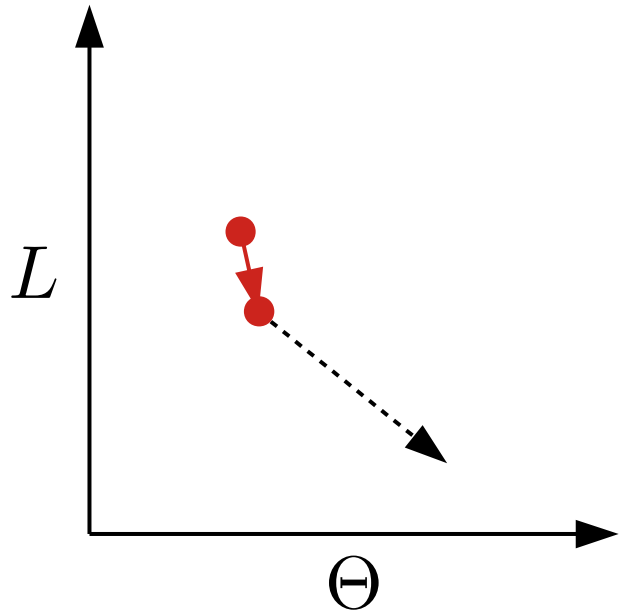
- Setting: unknown relation between loss L and parameters Θ
- Task: find Θ that minimize L given gradient $\frac{\partial L(\mathbf{y}_i, \hat{\mathbf{y}}_i)}{\partial \Theta}$



1. Initialize Θ randomly
2. Compute gradient
3. Move Θ a small step in direction of gradient
4. If not good enough: goto step 2

Stochastic Gradient Descent (SGD)

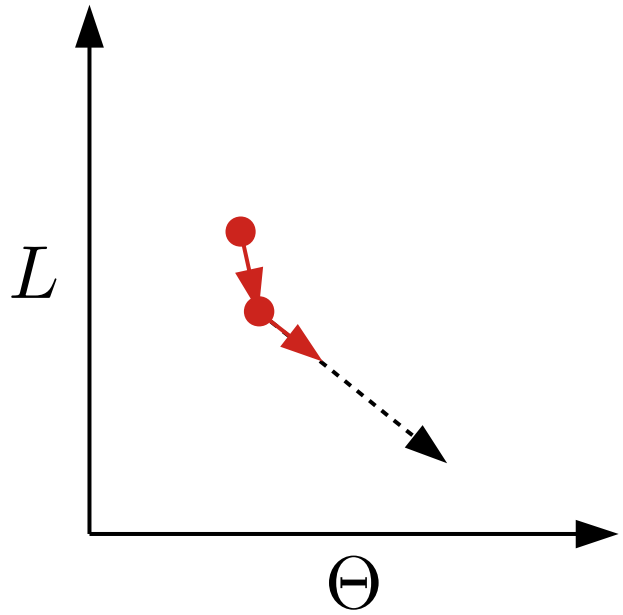
- Setting: unknown relation between loss L and parameters Θ
- Task: find Θ that minimize L given gradient $\frac{\partial L(\mathbf{y}_i, \hat{\mathbf{y}}_i)}{\partial \Theta}$



1. Initialize Θ randomly
2. Compute gradient
3. Move Θ a small step in direction of gradient
4. If not good enough: goto step 2

Stochastic Gradient Descent (SGD)

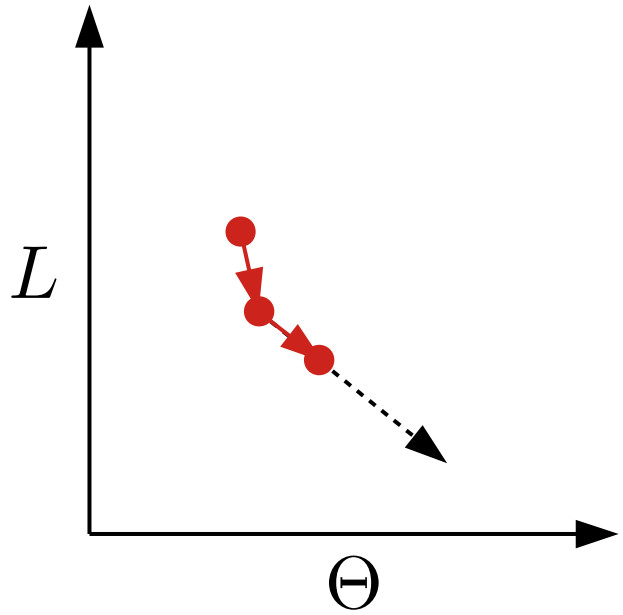
- Setting: unknown relation between loss L and parameters Θ
- Task: find Θ that minimize L given gradient $\frac{\partial L(\mathbf{y}_i, \hat{\mathbf{y}}_i)}{\partial \Theta}$



1. Initialize Θ randomly
2. Compute gradient
3. Move Θ a small step in direction of gradient
4. If not good enough: goto step 2

Stochastic Gradient Descent (SGD)

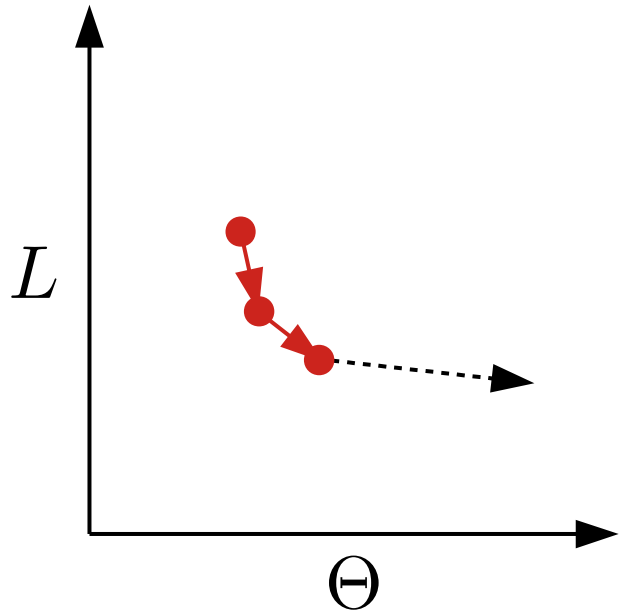
- Setting: unknown relation between loss L and parameters Θ
- Task: find Θ that minimize L given gradient $\frac{\partial L(\mathbf{y}_i, \hat{\mathbf{y}}_i)}{\partial \Theta}$



1. Initialize Θ randomly
2. Compute gradient
3. Move Θ a small step in direction of gradient
4. If not good enough: goto step 2

Stochastic Gradient Descent (SGD)

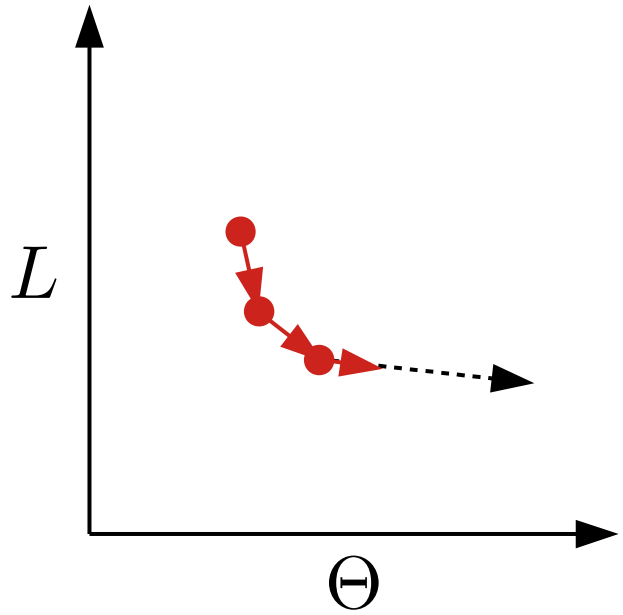
- Setting: unknown relation between loss L and parameters Θ
- Task: find Θ that minimize L given gradient $\frac{\partial L(\mathbf{y}_i, \hat{\mathbf{y}}_i)}{\partial \Theta}$



1. Initialize Θ randomly
2. Compute gradient
3. Move Θ a small step in direction of gradient
4. If not good enough: goto step 2

Stochastic Gradient Descent (SGD)

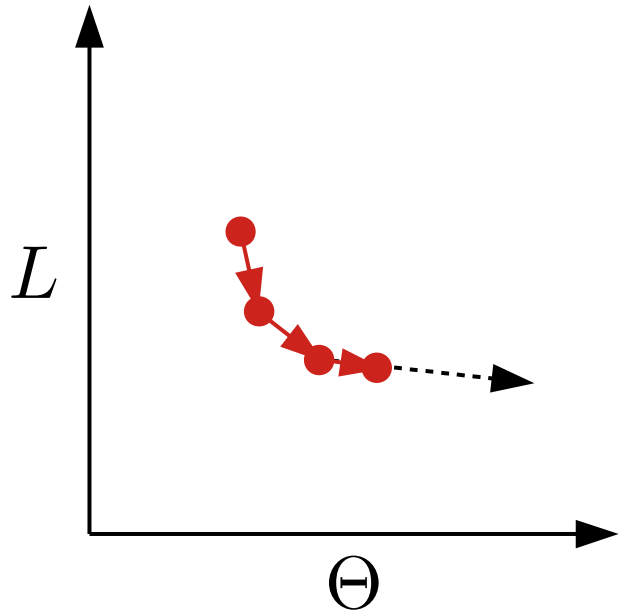
- Setting: unknown relation between loss L and parameters Θ
- Task: find Θ that minimize L given gradient $\frac{\partial L(\mathbf{y}_i, \hat{\mathbf{y}}_i)}{\partial \Theta}$



1. Initialize Θ randomly
2. Compute gradient
3. Move Θ a small step in direction of gradient
4. If not good enough: goto step 2

Stochastic Gradient Descent (SGD)

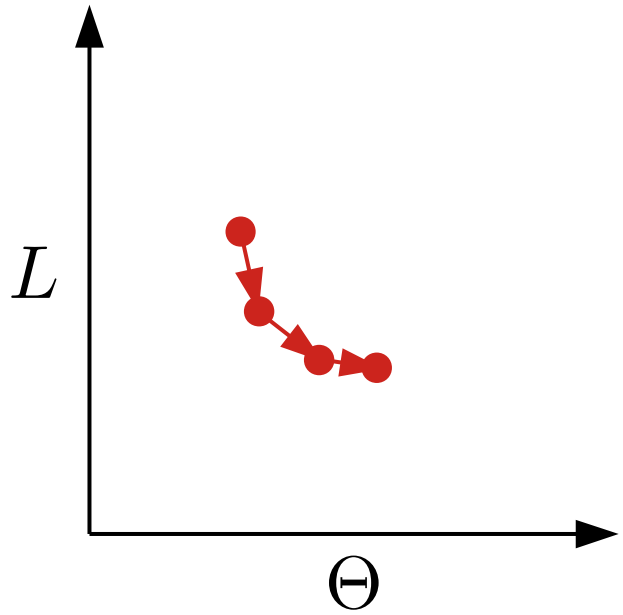
- Setting: unknown relation between loss L and parameters Θ
- Task: find Θ that minimize L given gradient $\frac{\partial L(\mathbf{y}_i, \hat{\mathbf{y}}_i)}{\partial \Theta}$



1. Initialize Θ randomly
2. Compute gradient
3. Move Θ a small step in direction of gradient
4. If not good enough: goto step 2
5. Else: done

Stochastic Gradient Descent (SGD)

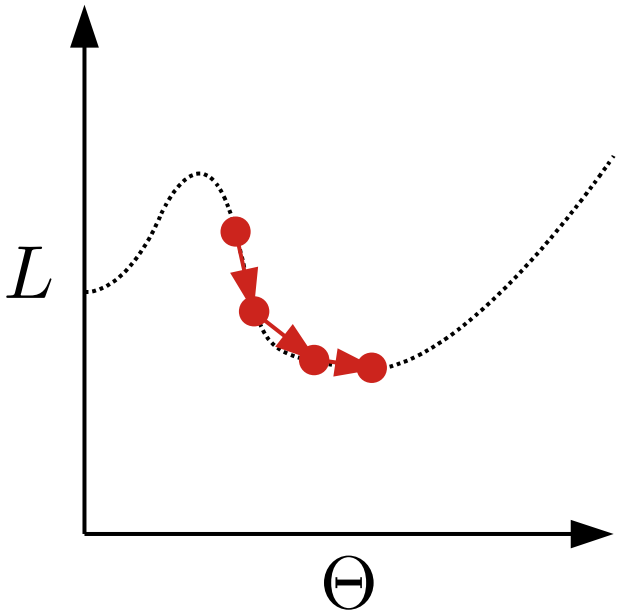
- Setting: unknown relation between loss L and parameters Θ
- Task: find Θ that minimize L given gradient $\frac{\partial L(\mathbf{y}_i, \hat{\mathbf{y}}_i)}{\partial \Theta}$



1. Initialize Θ randomly
2. Compute gradient
3. Move Θ a small step in direction of gradient
4. If not good enough: goto step 2
5. Else: done

Stochastic Gradient Descent (SGD)

- Setting: unknown relation between loss L and parameters Θ
- Task: find Θ that minimize L given gradient $\frac{\partial L(\mathbf{y}_i, \hat{\mathbf{y}}_i)}{\partial \Theta}$



1. Initialize Θ randomly
2. Compute gradient
3. Move Θ a small step in direction of gradient
4. If not good enough: goto step 2
5. Else: done

(Actual loss function for this example)

Stochastic Gradient Descent (SGD)

Complete Algorithm:

$\Theta \leftarrow$ random values

for given number of epochs:

shuffle dataset D

for each $(\mathbf{x}_i, \mathbf{y}_i) \in D$:

$$\Theta \leftarrow \Theta - \eta \cdot \frac{\partial L(\mathbf{y}_i, f(\mathbf{x}_i; \Theta))}{\partial \Theta}$$

return Θ

Stochastic Gradient Descent (SGD)

Complete Algorithm:

$\Theta \leftarrow$ random values

for given number of epochs:

  shuffle dataset D

  for each $(\mathbf{x}_i, \mathbf{y}_i) \in D$:

$$\Theta \leftarrow \Theta - \eta \cdot \frac{\partial L(\mathbf{y}_i, f(\mathbf{x}_i; \Theta))}{\partial \Theta}$$

return Θ

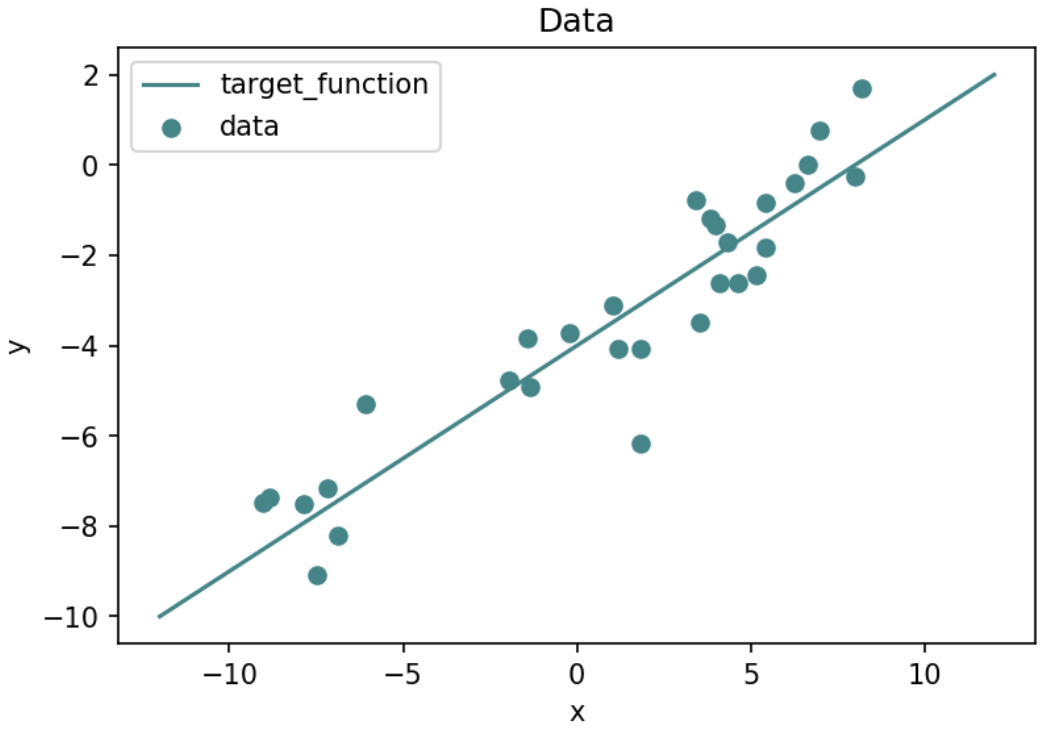
Learning rate (step size)

Learning Rate?

- Learning rate is a **hyperparameter** (parameter not estimated through training)
- **Learning rate (and other hyperparameters) need to be tuned manually for every learning problem!**
- Learning rate controls speed until convergence
- If learning rate is high: faster training, but might skip over narrow pits
- If learning rate is low: slower training, but might be unable to climb steep mountains

Dataset: Points from Line + Error

- Artificial target function: $f_{\text{target}}(x) = x \cdot 0.5 - 4$
- Same structure as our model function (i.e. learnable)
- Data points $(x_i, y_i) \in D$: 30 points randomly sampled from function + random gaussian noise



Ex 1: Simple Linear Regression

- Code: `01_simple_linear_regression__disturbed_line.ipynb`
- Dataset: points from straight line + random error
- Just two scalar parameters: $\Theta = \{m, b\}$
- Model function: $\hat{y}_i = f(x_i; \Theta) = x_i \cdot m + b$
- Loss: squared error $L(y_i, \hat{y}_i) = (y_i - \hat{y}_i)^2$
- Training algorithm: stochastic gradient descent

Ex 1: Imports

```
1 import numpy as np
2 import tensorflow as tf
```

- Aliases for required libraries
- NumPy: defacto standard math library for Python
- TensorFlow: machine learning library

Ex 1: Dataset (Points from Line + Error)

```python
def target_function(x):
    return x * 0.5 - 4


num_samples = 30
# Randomly sampled values in [-10, 10]
xs = np.random.uniform(low=-10, high=10, size=num_samples)
# Intended target value plus random noise
ys = target_function(xs) + np.random.normal(loc=0, scale=1, size=num_samples)

data = np.array(list(zip(xs, ys)))
```

- `zip()` takes two lists $[a_1, a_2, \dots], [b_1, b_2, \dots]$ and yields $(a_1, b_1), (a_2, b_2), \dots$
- `data` looks like this:

```
[[ 1.03 -3.11]
 [ 1.17 -4.06]
 [ 6.28 -0.39]
 ...
 [ 8.02 -0.24]]
```

Ex 1: Model + Loss + Training Algorithm

```
# Hyperparameters
learning_rate = 0.005
num_epochs = 20

# Model Definition
x = tf.placeholder(tf.float32)
y = tf.placeholder(tf.float32)

m = tf.Variable(1.0)
b = tf.Variable(0.0)

y_prediction = x * m + b

loss = (y - y_prediction) ** 2
train_op = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)
```

- `tf.placeholder()` defines inputs to our model that will be filled later
- `tf.Variable()` defines parameters with their initial value that will be adjusted during training

Ex 1: Model + Loss + Training Algorithm

```
1  # Hyperparameters
2  learning_rate = 0.005
3  num_epochs = 20
4
5  # Model Definition
6  x = tf.placeholder(tf.float32)
7  y = tf.placeholder(tf.float32)
8
9  m = tf.Variable(1.0)
10 b = tf.Variable(0.0)
11
12 y_prediction = x * m + b
13
14 loss = (y - y_prediction) ** 2
15 train_op = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)
```

Promise: we will go over the types of these expressions in a second. For now, observe the similarity to the mathematical specification.

- `tf.placeholder()` defines inputs to our model that will be filled later
- `tf.Variable()` defines parameters with their initial value that will be adjusted during training

Ex 1: Training the Model

```
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())

    # Training
    for epoch in range(num_epochs):
        np.random.shuffle(data)
        for _x, _y in data:
            _loss, _train_op = sess.run(
                (loss, train_op), feed_dict={x: _x, y: _y})
```

- Session object `sess` keeps track of internal `tf.Variable()` state
- `x`, `y`, `loss`, and `train_op` are directly from our model code
- `feed_dict` contains values for `tf.placeholder()`s
- My code convention: values for/from the model take underscore prefix (model: `x` → value: `_x`)

Ex 1: Training the Model

```
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())

    # Training
    for epoch in range(num_epochs):
        np.random.shuffle(data)
        for _x, _y in data:
            _loss, _train_op = sess.run(
                (loss, train_op), feed_dict={x: _x, y: _y})
```

- Compare to algorithm in pseudocode:
- One epoch = iterating over dataset once

$\Theta \leftarrow$ random values

for given number of epochs:

shuffle dataset D

for each $(\mathbf{x}_i, \mathbf{y}_i) \in D$:

$$\Theta \leftarrow \Theta - \eta \cdot \frac{\partial L(\mathbf{y}_i, f(\mathbf{x}_i; \Theta))}{\partial \Theta}$$

return Θ

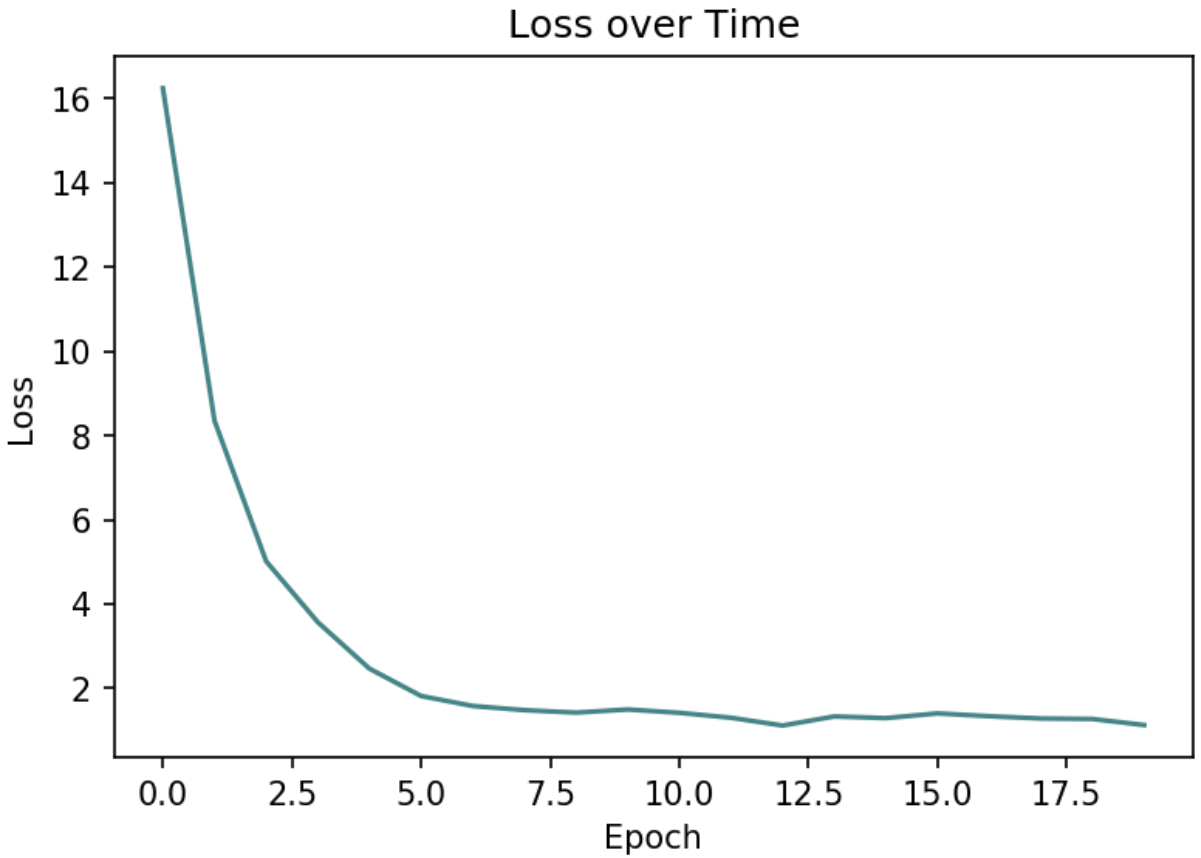
Ex 1: Training the Model

» If we print `epoch` and average `loss` for each iteration:

```
Epoch:  1, Loss: 16.24191335042318
Epoch:  2, Loss:  8.34964171623190
Epoch:  3, Loss:  5.01514179315418
Epoch:  4, Loss:  3.56995424857401
Epoch:  5, Loss:  2.46738942265510
Epoch:  6, Loss:  1.81869741876920
Epoch:  7, Loss:  1.58143185730829
Epoch:  8, Loss:  1.48245865919549
Epoch:  9, Loss:  1.42496345573369
Epoch: 10, Loss:  1.49863767982848
Epoch: 11, Loss:  1.41974687864033
Epoch: 12, Loss:  1.30234955443690
Epoch: 13, Loss:  1.11769525284374
Epoch: 14, Loss:  1.33366958821813
Epoch: 15, Loss:  1.29429860597010
Epoch: 16, Loss:  1.40585160747868
Epoch: 17, Loss:  1.33890924156488
Epoch: 18, Loss:  1.28456278786373
Epoch: 19, Loss:  1.27327905286025
Epoch: 20, Loss:  1.12928571269537
```

Ex 1: Training the Model

» As a plot:



» Can observe **loss convergence** around epoch 6

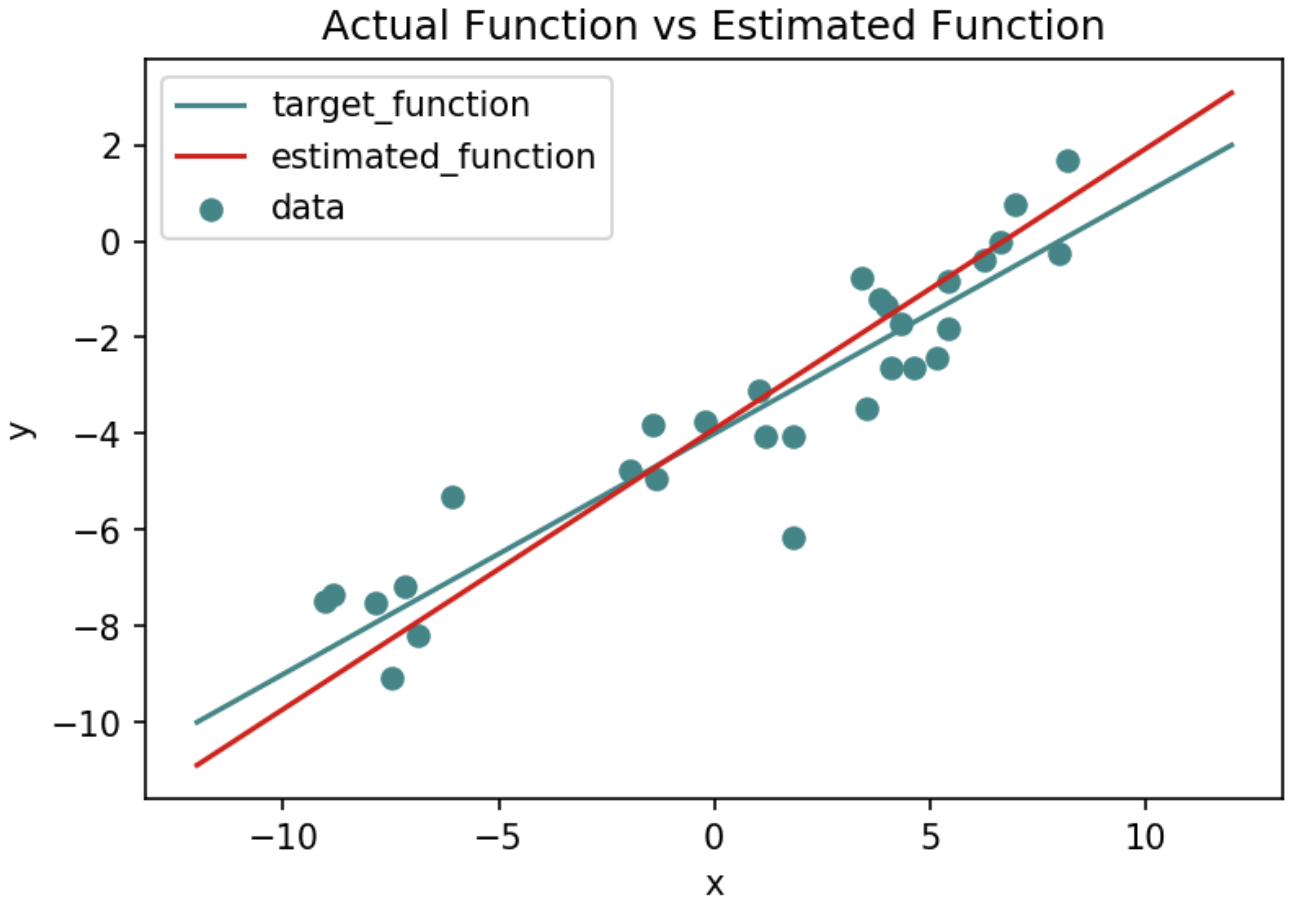
Ex 1: Using the Trained Model

```
    # Introspection
    _m, _b = sess.run([m, b])
    print('Estimated m:', _m)
    print('Estimated b:', _b)

    # Prediction
    ys_actual = []
    ys_predicted = []
    for _x, _y in data:
        ys_actual.append(_y)
        ys_predicted.append(sess.run(y_prediction, feed_dict={x: _x}))
```

- Same `sess` object as before
- Estimated parameters:

```
Estimated m: 0.5824312
Estimated b: -3.903798
```

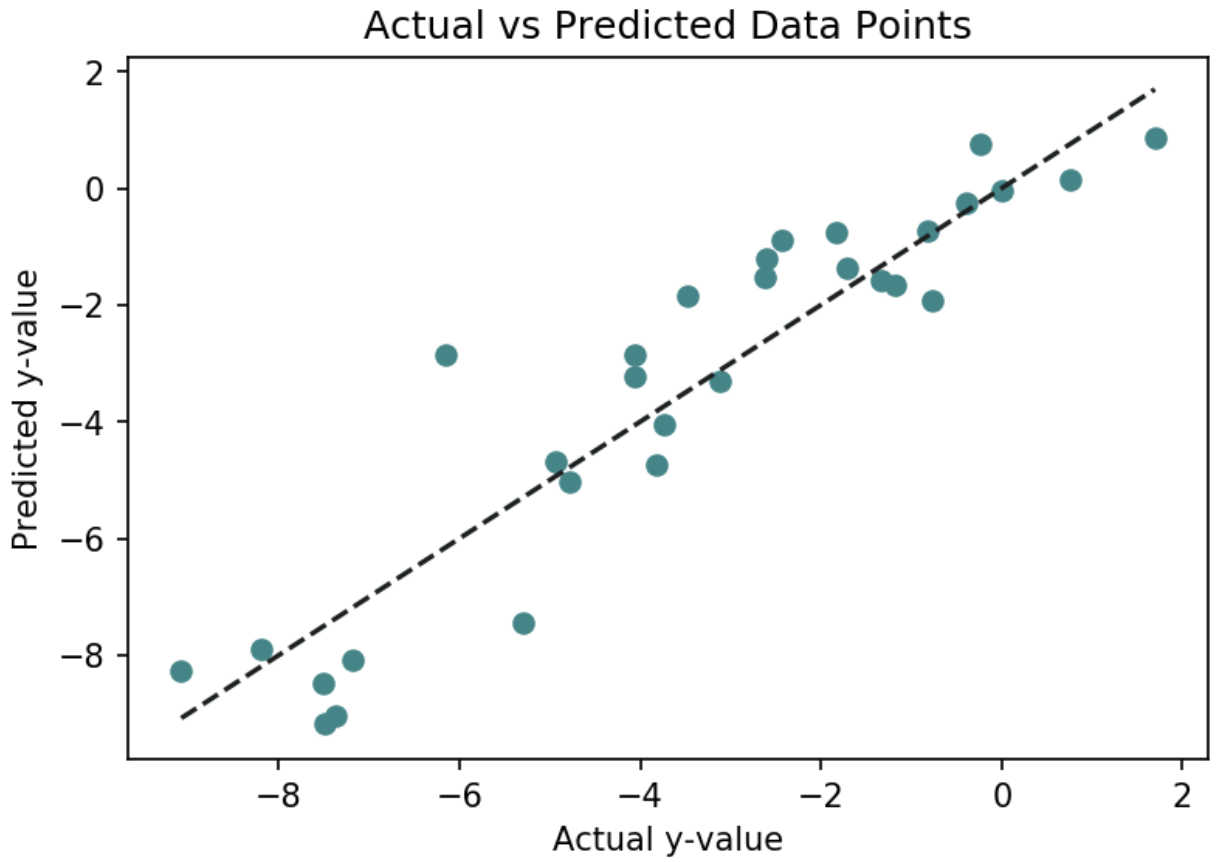
Ex 1: Actual vs Estimated Function



- Estimated function is pretty close to actual function
- More data samples would probably improve model

Ex 1: Actual vs Predicted Data Points

» Another useful diagram type is difference from actual to predicted output values



Ok, what is actually happening here?

- Where do we calculate the gradient $\frac{\partial L(\mathbf{y}_i, \hat{\mathbf{y}}_i)}{\partial \Theta}$?
- Where do we update our model parameters `m` and `b`?
- Why are there model variables `x` and normal variables `_x`?

Ok, what is actually happening here?

- Where do we calculate the gradient $\frac{\partial L(\mathbf{y}_i, \hat{\mathbf{y}}_i)}{\partial \Theta}$?
- Where do we update our model parameters `m` and `b`?
- Why are there model variables `x` and normal variables `_x`?

Answer

- Most of the magic is hidden in this line:

```
1 train_op = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)
```

- We used TensorFlow's computation graph abstraction

Computation Graph Abstraction

- The computation graph is a representation of arbitrary mathemetical computation (specified in our code)
- Allows for **automatic differentiation** by knowing the partial derivative of each node and using chain rule

Computation Graph Abstraction

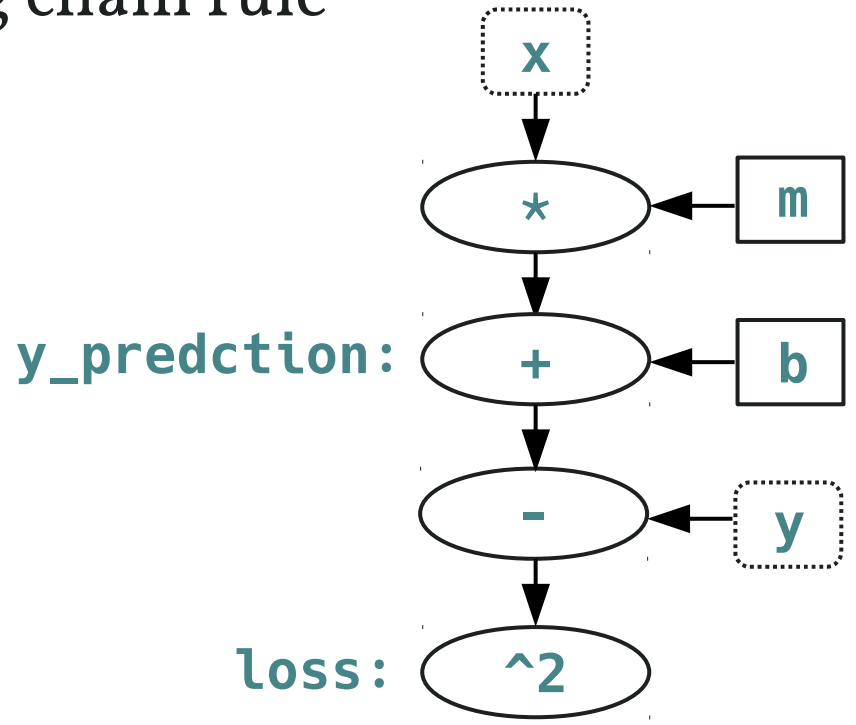
- The computation graph is a representation of arbitrary mathemetical computation (specified in our code)
- Allows for **automatic differentiation** by knowing the partial derivative of each node and using chain rule

```
1 x = tf.placeholder(tf.float32)
2 y = tf.placeholder(tf.float32)
3
4 m = tf.Variable(1.0)
5 b = tf.Variable(0.0)
6
7 y_prediction = x * m + b
8
9 loss = (y - y_prediction) ** 2
```

Computation Graph Abstraction

- The computation graph is a representation of arbitrary mathemetical computation (specified in our code)
- Allows for **automatic differentiation** by knowing the partial derivative of each node and using chain rule

```
1 x = tf.placeholder(tf.float32)
2 y = tf.placeholder(tf.float32)
3
4 m = tf.Variable(1.0)
5 b = tf.Variable(0.0)
6
7 y_prediction = x * m + b
8
9 loss = (y - y_prediction) ** 2
```



Legend: `tf.placeholder()` `tf.Variable()` `math op`

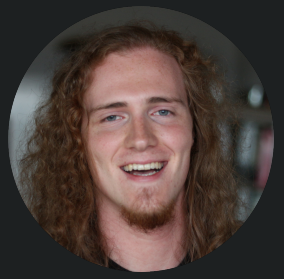
TensorFlow Magic

```
1  x = tf.placeholder(tf.float32)
2  y = tf.placeholder(tf.float32)
3
4  m = tf.Variable(1.0)
5  b = tf.Variable(0.0)
6
7  y_prediction = x * m + b
8
9  loss = (y - y_prediction) ** 2
10 train_op = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)
```

- Internal: operators like `+` and `*` overloaded on placeholders and variables, actual functions: `tf.add()` and `tf.multiply()`
- TensorFlow knows derivation for all internal functions
- On each evaluation of `train_op` the current gradients are calculated and variables are moved according to learning rate

Summary: Simple Linear Regression

- Simple Linear Regression:
  - Both input and output are scalar
  - Model function: $\hat{y}_i = f(x_i; \Theta) = x_i \cdot m + b$
- Stochastic Gradient Descent
  - Algorithm to find optimal parameters Θ
- Computation Graph Abstraction
  - Allows for automatic differentiation of model functions



LUKASSCHMELZEISEN.

lukas@uni-koblenz.de
lschmelzeisen.com

24 Sep 2018

Introduction to Supervised Learning with TensorFlow

» Linear Regression

Linear Regression

- Applicable if input is a vector and output is scalar:
  - $\mathbf{x}_i = \left[x_{i1} \dots x_{id_{\text{in}}}\right] \in \mathbb{R}^{d_{\text{in}}}$ (for simple LR we had: $d_{\text{in}} = 1$)
  - $\mathbf{y}_i = y_i \in \mathbb{R}$
- Again, two parameters: $\Theta = \{\mathbf{w}, b\}$
  - weight vector $\mathbf{w} = \left[w_1 \dots w_{d_{\text{in}}}\right]$
  - bias b
- Model function: $\hat{y}_i = f(\mathbf{x}_i; \Theta) = \mathbf{x}_i \mathbf{w}^\intercal + b$
  (using dot product: $\mathbf{x}_i \mathbf{w}^\intercal = \sum_{j=1}^{d_{\text{in}}} x_{ij} w_j$)
- Loss is still **squared error**: $L(y_i, \hat{y}_i) = (y_i - \hat{y}_i)^2$

Dataset: Boston Housing

- Originally published in Harrison & Rubinfeld (1978).
- Download: https://www.cs.toronto.edu/~delve/data/boston/bostonDetail.html
- Collection of median house prices from the Boston area for 13 input variables (506 data points):
  - *CRIM: per capita crime rate by town*
  - *AGE: proportion of owner-occupied units built prior to 1940*
  - *INDUS: proportion of non-retail business acres per town*
  - *RM: average number of rooms*
  - …

Harrison & Rubinfeld (1978). "Hedonic Housing Prices and the Demand for Clean Air". *J. Environ. Economics & Management*.

Dataset: Boston Housing

» Example:

```
  CRIM  ZN    INDUS     B     LSTAT    Y
[[23.58 24.28 22.71 ... 22.54 24.44    22.92]
 [21.18 21.11 21.01 ... 21.3  22.04    21.11]
 [34.28 34.21 34.11 ... 34.4  35.1     33.49]
 ...
 [23.49 23.41 24.02 ... 25.08 24.34    22.92]
 [21.59 21.51 22.12 ... 23.18 22.4     21.13]
 [11.49 11.41 12.02 ... 13.08 12.34    11.23]]
```

Train/Test Split

- In machine learning, we usually don't care about how well we can fit a function to our dataset
- Being able to correctly predict unseen data is more important!
- Therefore, we split our dataset randomly into **training** and **testing** subsets
  - Training data is used to find best values for parameters
  - Testing data is used to evaluate quality of estimated model Θ
  - In this talk we will always use 60% of data for training and the remaining 40% for testing

Train/Test Split: Implementation

```
boston_housing = sklearn.datasets.load_boston()
xs = boston_housing.data
ys = boston_housing.target

data = list(zip(xs, ys))

# Perform 60%/40% training/test split
split_index = int(len(data) * 0.6)
train_data = data[:split_index]
test_data = data[split_index:]
```

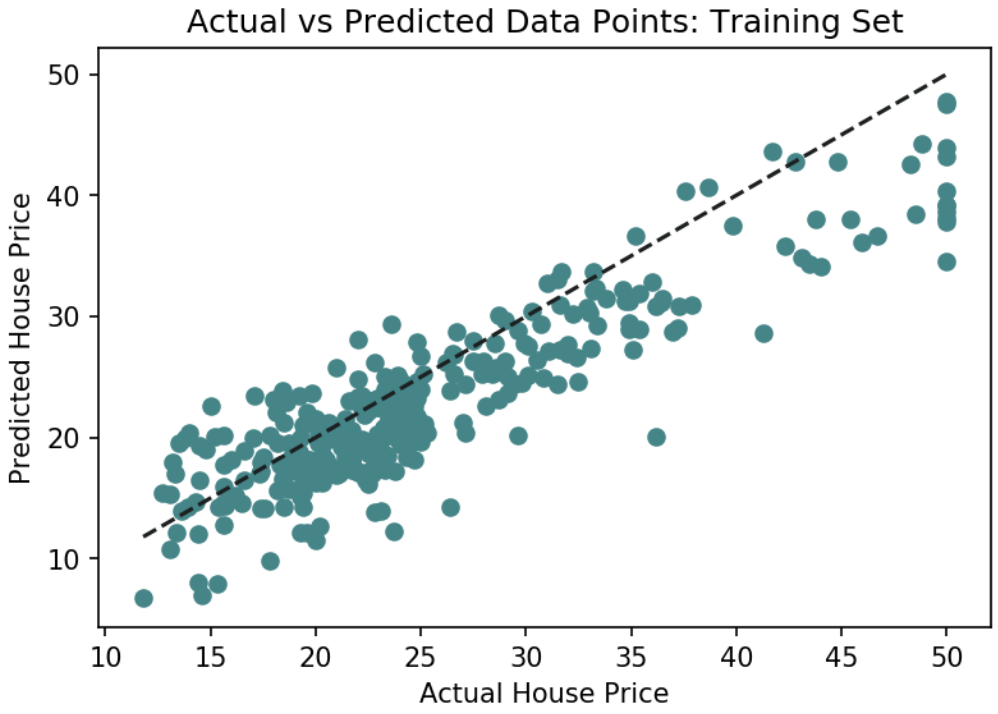
- Scikit-learn library provides helper to load dataset
- Splitting the dataset uses Python's list slicing

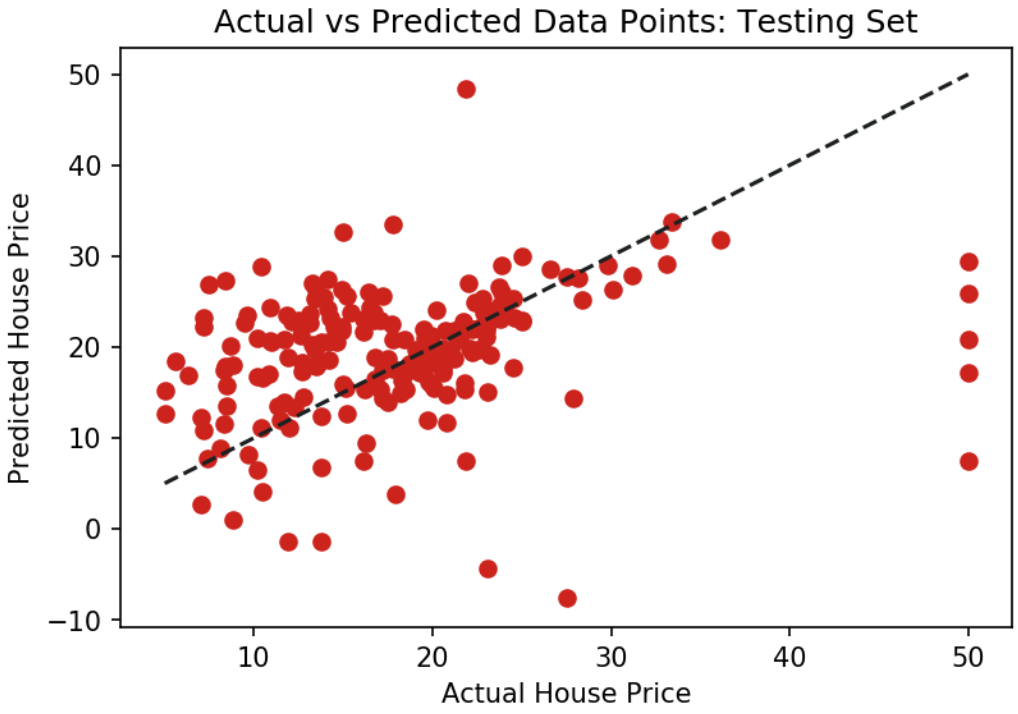
Ex 2: Simple Linear Regression (Housing)

- Code: `02_simple_linear_regression__housing.ipynb`
- For comparision, we will first run simple linear regression on our dataset
- Simple LR only allows for one input variable, let's chose:
  - *RM: average number of rooms* (highest correlation with output)
- Code identical to Ex 1 besides dataset

Ex 2: Results

- Evaluation metric: mean squared error (lower is better)
  - Training data: **21.25**
  - Testing data: **82.42**





Ex 3: Linear Regression (Housing)

- Code: `03_linear_regression__housing.ipynb`
- Now with "normal" linear regression:

$$\hat{y}_i = f(\mathbf{x}_i; \Theta) = \mathbf{x}_i \mathbf{w}^\intercal + b$$

- Code identical to Ex 2, besides:

```
# Hyperparameters
learning_rate = 0.000001
num_epochs = 500
num_features = len(train_data[0][0])

# Model Definition
x = tf.placeholder(tf.float32, shape=[num_features])
y = tf.placeholder(tf.float32)

w = tf.Variable(tf.random_normal(shape=[num_features], mean=0, stddev=1))
b = tf.Variable(0.0)

y_prediction = tf.tensordot(x, w, 1) + b

loss = (y - y_prediction) ** 2
train_op = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)
```

Ex 3: Linear Regression (Housing)

- Code: `03_linear_regression__housing.ipynb`
- Now with "normal" linear regression:

$$\hat{y}_i = f(\mathbf{x}_i; \Theta) = \mathbf{x}_i \mathbf{w}^\intercal + b$$

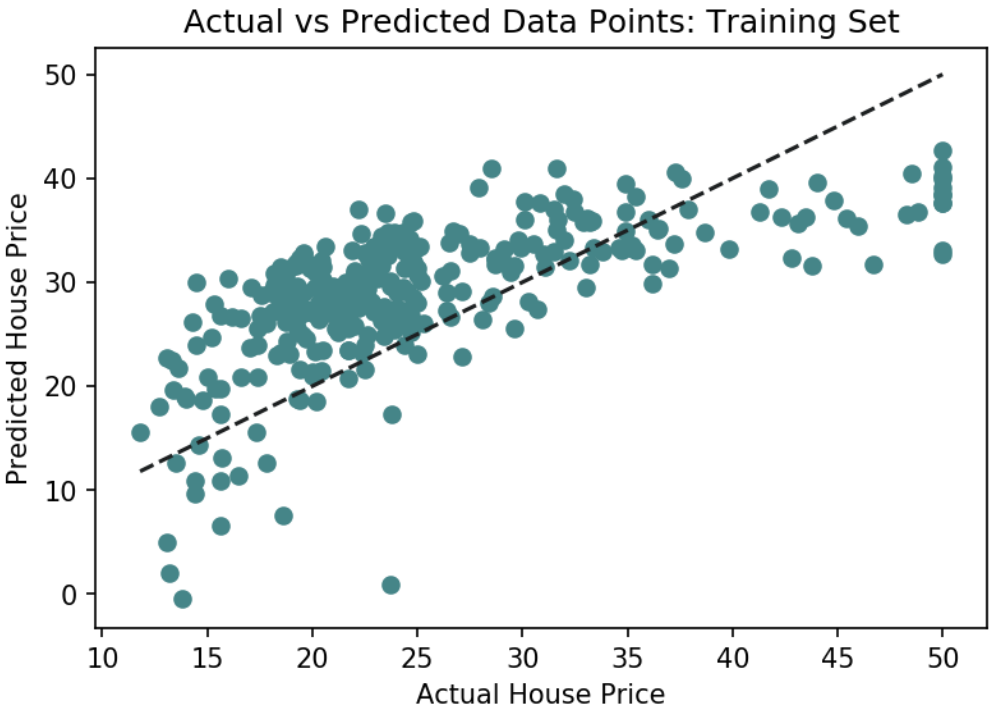
- Code identical to Ex 2, besides:

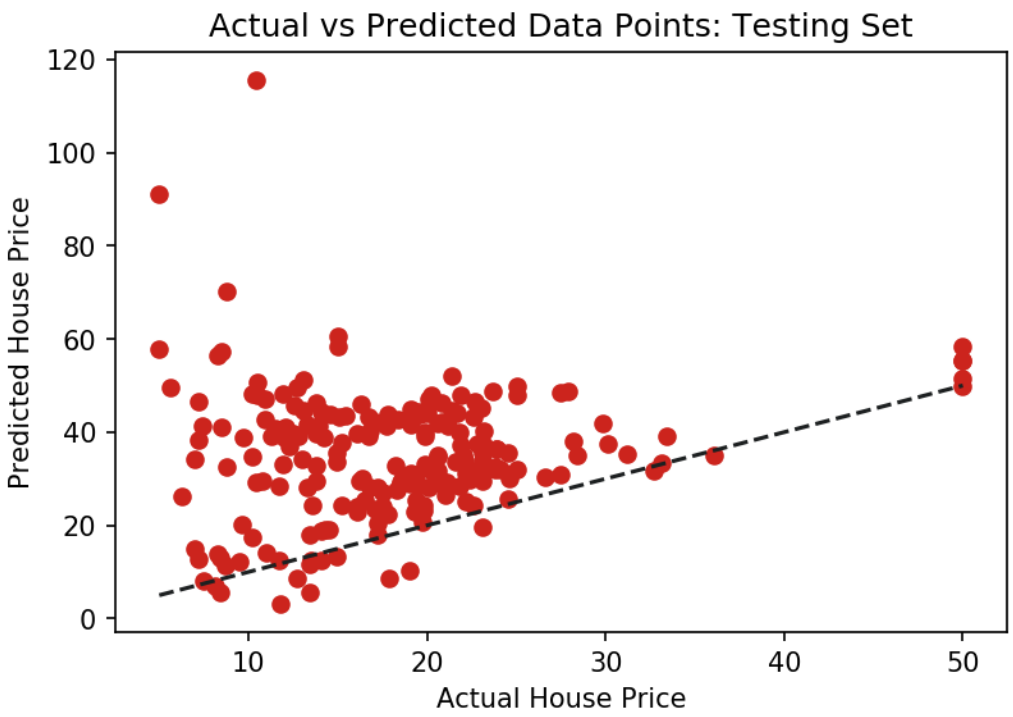
`x` and `w` are now vectors with `num_features` many dimensions!

```
# Hyperparameters
learning_rate = 0.000001
num_epochs = 500
num_features = len(train_data[0][0])

# Model Definition
x = tf.placeholder(tf.float32, shape=[num_features])
y = tf.placeholder(tf.float32)

w = tf.Variable(tf.random_normal(shape=[num_features], mean=0, stddev=1))
b = tf.Variable(0.0)

y_prediction = tf.tensordot(x, w, 1) + b

loss = (y - y_prediction) ** 2
train_op = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)
```

Ex 3: Results

- Mean squared error (lower is better)
  - Training data: **56.76** *(was **21.25** for simple LR)*
  - Testing data: **500.43** *(was **82.42** for simple LR)*





Wait what?

- Why did our results become so much worse?
- Didn't we add more input variables, i.e. more data to predict from?

Wait what?

- Why did our results become so much worse?
- Didn't we add more input variables, i.e. more data to predict from?

Answer

- Probably because the magnitude of all input variables varies a lot!
- Solution: standardization of input data

Ex 4: Standardization

- Code: `04_linear_regression__housing__with_standardization.ipynb`

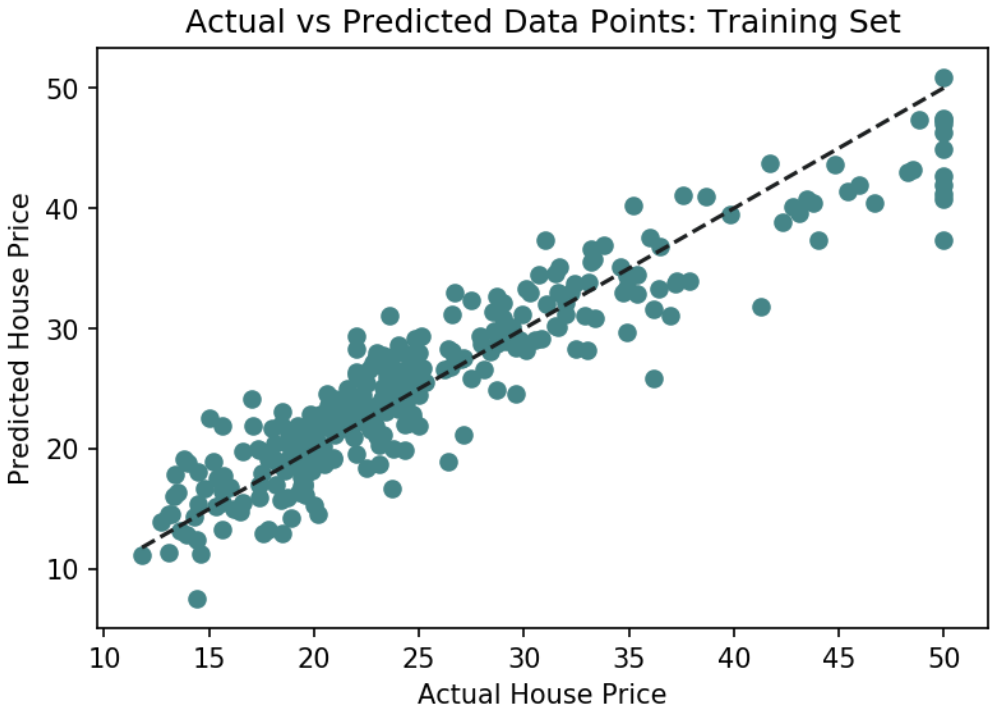
```
1 xs = (xs - np.mean(xs, axis=0)) / np.std(xs, axis=0)
```

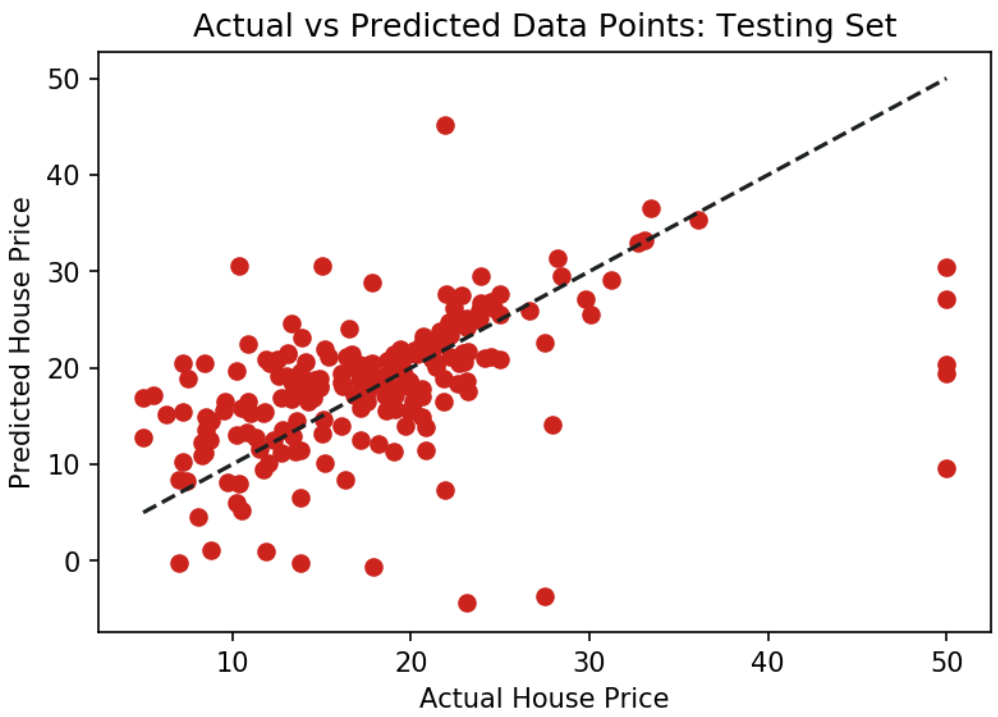
(Some people call this normalization)

- For each feature:
  - Subtract by population mean and devide by standard deviation
  - Resulting value are **standard scores** (also called **z-values**) that indicate how many standard deviations the original value was away from the mean
- From now on, automatically applied to all further experiments

Ex 4: Results (with Standardization)

- Mean squared error (lower is better)
  - Training data: **10.23** *(was* ***56.76*** *unstandardized,* ***21.25*** *for simple LR)*
  - Testing data: **60.92** *(was* ***500.43*** *unstandardized,* ***82.42*** *for simple LR)*





Tensors

- Tensors are a generalization of vectors and matrices, basically multidimensional arrays
- The **rank** of a tensor gives its number of dimensions
- The **shape** of a tensor gives the number of elements in each dimension

Name	Rank	Example Shape	Exampe Value(s)
Scalar	0	()	5
Vector	1	(3)	[3, 4, -2]
Matrix	2	(4, 2)	[[7, 3], [5, 2], [1, 1], [4, 9]]
3-Tensor	3	(3, 2, 2)	[[[2, 5], [1, 2]], [[3, 0], [4, 5]], [[7, 5], [4, 2]]]

...

Tensors

```
1  x = tf.placeholder(tf.float32, shape=[num_features])
2  y = tf.placeholder(tf.float32)
3
4  w = tf.Variable(tf.random_normal(shape=[num_features], mean=0, stddev=1))
5  b = tf.Variable(0.0)
6
7  y_prediction = tf.tensordot(x, w, 1) + b
8
9  loss = (y - y_prediction) ** 2
10 train_op = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)
```

- In TensorFlow every variable and operation is a `tf.Tensor`
- Each `tf.Tensor` has a datatype like `tf.float32`, `tf.int32`, … (most of the time implicitly defined)
- Each `tf.Tensor` defines a partial computation that can be evaluated to a NumPy tensor value at runtime with `sess.run()`

Tensors

All functions return tensors!

```
1  x = tf.placeholder(tf.float32, shape=[num_features])
2  y = tf.placeholder(tf.float32)
3
4  w = tf.Variable(tf.random_normal(shape=[num_features], mean=0, stddev=1))
5  b = tf.Variable(0.0)
6
7  y_prediction = tf.tensordot(x, w, 1) + b
8
9  loss = (y - y_prediction) ** 2
10 train_op = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)
```

- In TensorFlow every variable and operation is a `tf.Tensor`
- Each `tf.Tensor` has a datatype like `tf.float32`, `tf.int32`, … (most of the time implicitly defined)
- Each `tf.Tensor` defines a partial computation that can be evaluated to a NumPy tensor value at runtime with `sess.run()`

Mini-Batch Gradient Descent

- Normal stochastic gradient descent updates parameters after every training example
- Improving parameters for one specific example always has the risk of making things worse for other examples
- Computation of each training example can only be performed once the parameter update of the previous one is completed
- Mini-batch gradient descent:
  - Compute gradients of a small batch of training examples in parallel
  - Update parameters with average of batch gradients
- Results in great speed up in training

Mini-Batch Gradient Descent

$\Theta \leftarrow$ random values

for given number of epochs:

sample a batch B of examples from D

$\hat{\mathbf{g}} \leftarrow \mathbf{0}$

for each $(\mathbf{x}_i, \mathbf{y}_i) \in B$:

$$\hat{\mathbf{g}} \leftarrow \hat{\mathbf{g}} + \frac{1}{|B|} \frac{\partial L(\mathbf{y}_i, f(\mathbf{x}_i; \Theta))}{\partial \Theta}$$

$\Theta \leftarrow \Theta - \eta \cdot \hat{\mathbf{g}}$

return Θ

Compare: Stochastic Gradient Descent

$\Theta \leftarrow$ random values

for given number of epochs:

shuffle dataset D

for each $(\mathbf{x}_i, \mathbf{y}_i) \in D$:

$$\Theta \leftarrow \Theta - \eta \cdot \frac{\partial L(\mathbf{y}_i, f(\mathbf{x}_i; \Theta))}{\partial \Theta}$$

return Θ

Ex 5: Linear Regression (Batching)

- Code: `05_linear_regression__housing__with_batching.ipynb`
- Very similar to Ex 4, but to implement batching we need to change
  - Model definition
  - Model training

Compare: Model Definition before

```
# Hyperparameters
learning_rate = 0.000001
num_epochs = 500
num_features = len(train_data[0][0])

# Model Definition
x = tf.placeholder(tf.float32, shape=[num_features])
y = tf.placeholder(tf.float32)

w = tf.Variable(tf.random_normal(shape=[num_features], mean=0, stddev=1))
b = tf.Variable(0.0)

y_prediction = tf.tensordot(x, w, 1) + b

loss = (y - y_prediction) ** 2
train_op = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)
```

Ex 5: Model Definition with Batching

```
batch_x = tf.placeholder(tf.float32, shape=[None, num_features])
batch_y = tf.placeholder(tf.float32, shape=[None])

w = tf.Variable(tf.random_normal(shape=[num_features], mean=0, stddev=1))
b = tf.Variable(0.0)

y_prediction = tf.tensordot(batch_x, w, 1) + b

loss = tf.reduce_mean((batch_y - y_prediction) ** 2)
train_op = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)
```

- Now: matrix of features `batch_x` and vector of targets `batch_y`
- `None` as dimension means that dimensionality can vary during runtime
- All operations are still well defined

Ex 5: Model Definition with Batching

```
1  batch_x = tf.placeholder(tf.float32, shape=[None, num_features])
2  batch_y = tf.placeholder(tf.float32, shape=[None])
3
4  w = tf.Variable(tf.random_normal(shape=[num_features], mean=0, stddev=1))
5  b = tf.Variable(0.0)
6
7  y_prediction = tf.tensordot(batch_x, w, 1) + b
8
9  loss = tf.reduce_mean((batch_y - y_prediction) ** 2)
10 train_op = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)
```

- Now: matrix of features `batch_x` and vector of targets `batch_y`
- `None` as dimension means that dimensionality can vary during runtime
- All operations are still well defined

Exact same code as before, but now a vector!
(different input shapes, batching increases rank by one)

Ex 5: Training the Model

```
batch_size = 10

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())

    # Training
    for epoch in range(num_epochs):
        np.random.shuffle(train_data)
        for i in range(0, len(train_data), batch_size):
            _batch_x, _batch_y = zip(*train_data[i:i + batch_size])
            _loss, _train_op = sess.run(
                (loss, train_op),
                feed_dict={batch_x: _batch_x, batch_y: _batch_y})
```

- Batch selection is a bit hidden in Python magic
- `_batch_x` and `_batch_y` go through training data
  10 examples at a time
- Everything else still works the same

Ex 5: Training Time

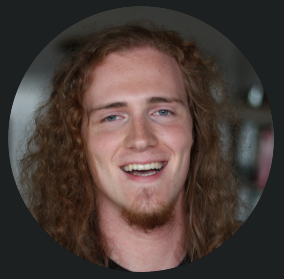
- Ex 4: stochastic gradient descent
  100 epochs: **10.34** seconds
- Ex 5: batch gradient descent (`batch_size = 10`)
  500 epochs: **7.08** seconds

Remark for Reading ML-Papers

- We have seen that the actual model definition code stays the same with batching
- The same is true for formulas: $\hat{y}_i = f(\mathbf{x}_i; \Theta) = \mathbf{x}_i \mathbf{w}^\intercal + b$
  - Without batching: input $\mathbf{x}_i$ is a vector, output $\hat{y}_i$ is a scalar
  - With batching: input would become a matrix and output a vector
- Usually publications just present the non-batching formulas; generalizing to their batching equivalents is often left as an exercise to the reader (not always straightforward)
- Implementations are always with batching!

Summary: Linear Regression

- Linear Regression
  - Inputs are now vectors, outputs are still scalar
  - Model function: $\hat{y}_i = f(\mathbf{x}_i; \Theta) = \mathbf{x}_i \mathbf{w}^\intercal + b$
- Train/Test Split
- Tensors
  - Basically just multidimensional arrays
- Mini-Batch Gradient Descent
  - Allows to speed up training



LUKASSCHMELZEISEN.

lukas@uni-koblenz.de
lschmelzeisen.com

24 Sep 2018

Introduction to Supervised Learning with TensorFlow

» Logistic Regression

Logistic Regression

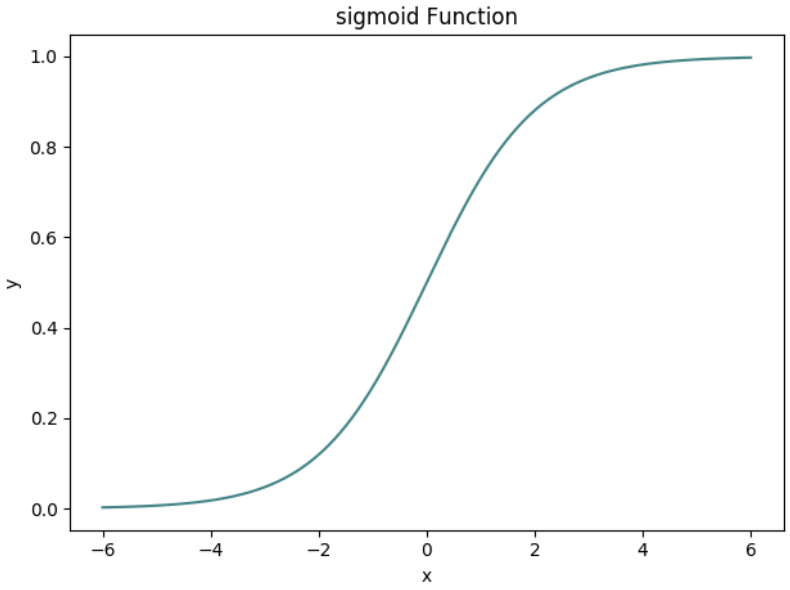
- Extension of linear regression to classification
- Misnomer: not a technique for regression problems!
- Applicable if input is a vector and output binary:
  - $\mathbf{x}_i = \left[x_{i1} \dots x_{id_{\text{in}}}\right] \in \mathbb{R}^{d_{\text{in}}}$
  - Only two different values for $\mathbf{y}_i$: $\{y_1, y_2\}$
- Output classes unsually encoded through `1` and `0`

Logistic Regression

- Same parameters as linear regression: $\Theta = \{\mathbf{w}, b\}$
- Model function:

$$P(\hat{\mathbf{y}}_i = y_1 \mid \mathbf{x}_i; \Theta) = \sigma(\mathbf{x}_i \mathbf{w}^\intercal + b)$$
$$P(\hat{\mathbf{y}}_i = y_2 \mid \mathbf{x}_i; \Theta) = 1 - P(\hat{\mathbf{y}}_i = y_1 \mid \mathbf{x}_i; \Theta)$$

- Uses the sigmoid function σ:
  - $\sigma(x) = \frac{1}{1+e^{-x}}$
  - Transforms values in $[-\infty, \infty]$ into values in $[0, 1]$, i.e. probabilities



Logistic Regression: Decision

- Model function:

$$P(\hat{\mathbf{y}}_i = y_1 \mid \mathbf{x}_i; \Theta) = \sigma(\mathbf{x}_i \mathbf{w}^\intercal + b)$$
$$P(\hat{\mathbf{y}}_i = y_2 \mid \mathbf{x}_i; \Theta) = 1 - P(\hat{\mathbf{y}}_i = y_1 \mid \mathbf{x}_i; \Theta)$$

- To decide output prediction:
  - If $P(\hat{\mathrm{y}}_i = y_1 \mid \mathbf{x}_i; \Theta) > 0.5$: predict $\hat{\mathbf{y}}_i = y_1$
  - Else: predict $\hat{\mathbf{y}}_i = y_2$
- Because $\sigma(x) > 0.5 \iff x > 0$ the following is equivalent:
  - If $\mathbf{x}_i \mathbf{w}^\intercal + b > 0$: predict $\hat{\mathbf{y}}_i = y_1$
  - Else: predict $\hat{\mathbf{y}}_i = y_2$

Logistic Regression: Decision

- Model function:

$$P(\hat{\mathbf{y}}_i = y_1 \mid \mathbf{x}_i; \Theta) = \sigma(\mathbf{x}_i \mathbf{w}^\intercal + b)$$
$$P(\hat{\mathbf{y}}_i = y_2 \mid \mathbf{x}_i; \Theta) = 1 - P(\hat{\mathbf{y}}_i = y_1 \mid \mathbf{x}_i; \Theta)$$

- To decide output prediction:
  - If $P(\hat{\mathbf{y}}_i = y_1 \mid \mathbf{x}_i; \Theta) > 0.5$: predict $\hat{\mathbf{y}}_i = y_1$
  - Else: predict $\hat{\mathbf{y}}_i = y_2$
- Because $\sigma(x) > 0.5 \iff x > 0$ the following is equivalent:
  - If $\mathbf{x}_i \mathbf{w}^\intercal + b > 0$: predict $\hat{\mathbf{y}}_i = y_1$
  - Else: predict $\hat{\mathbf{y}}_i = y_2$

Wait, that's just linear regression!
Why worry about the sigmoid?

Loss for Classification: Cross Entropy

- Also known as **negative log likelihood**
- Should be be better suited for classification then squared error
- For hard binary classification (classes are exclusive):

$$L(\mathbf{y}_i, \hat{\mathbf{y}}_i) = -\log P(\hat{\mathbf{y}}_i = \mathbf{y}_i \mid \mathbf{x}_i; \Theta)$$

Reminder: logistic regression model

$$P(\hat{\mathbf{y}}_i = y_1 \mid \mathbf{x}_i; \Theta) = \sigma(\mathbf{x}_i \mathbf{w}^\intercal + b)$$

$$P(\hat{\mathbf{y}}_i = y_2 \mid \mathbf{x}_i; \Theta) = 1 - P(\hat{\mathbf{y}}_i = y_1 \mid \mathbf{x}_i; \Theta)$$

Loss for Classification: Cross Entropy

- Also known as **negative log likelihood**
- Should be be better suited for classification then squared error
- For hard binary classification (classes are exclusive):

$$L(\mathbf{y}_i, \hat{\mathbf{y}}_i) = -\log P(\hat{\mathbf{y}}_i = \mathbf{y}_i \mid \mathbf{x}_i; \Theta)$$

$$= \begin{cases} -\log \sigma(\mathbf{x}_i \mathbf{w}^\intercal + b) & \text{if } \mathbf{y}_i = y_1 \\ -\log(1 - \sigma(\mathbf{x}_i \mathbf{w}^\intercal + b)) & \text{if } \mathbf{y}_i = y_2 \end{cases}$$

Reminder: logistic regression model

$$P(\hat{\mathbf{y}}_i = y_1 \mid \mathbf{x}_i; \Theta) = \sigma(\mathbf{x}_i \mathbf{w}^\intercal + b)$$

$$P(\hat{\mathbf{y}}_i = y_2 \mid \mathbf{x}_i; \Theta) = 1 - P(\hat{\mathbf{y}}_i = y_1 \mid \mathbf{x}_i; \Theta)$$

Loss for Classification: Cross Entropy

- Also known as **negative log likelihood**
- Should be be better suited for classification then squared error
- For hard binary classification (classes are exclusive):

$$L(\mathbf{y}_i, \hat{\mathbf{y}}_i) = -\log P(\hat{\mathbf{y}}_i = \mathbf{y}_i \mid \mathbf{x}_i; \Theta)$$

$$= \begin{cases} -\log \sigma(\mathbf{x}_i \mathbf{w}^\intercal + b) & \text{if } \mathbf{y}_i = y_1 \\ -\log(1 - \sigma(\mathbf{x}_i \mathbf{w}^\intercal + b)) & \text{if } \mathbf{y}_i = y_2 \end{cases}$$

$$= -\mathbf{y}_i \cdot \log \sigma(\mathbf{x}_i \mathbf{w}^\intercal + b) \quad \leftarrow \text{if } y_1 = 1 \wedge y_2 = 0$$

$$- (1 - \mathbf{y}_i) \cdot \log(1 - \sigma(\mathbf{x}_i \mathbf{w}^\intercal + b)$$

Reminder: logistic regression model

$$P(\hat{\mathbf{y}}_i = y_1 \mid \mathbf{x}_i; \Theta) = \sigma(\mathbf{x}_i \mathbf{w}^\intercal + b)$$

$$P(\hat{\mathbf{y}}_i = y_2 \mid \mathbf{x}_i; \Theta) = 1 - P(\hat{\mathbf{y}}_i = y_1 \mid \mathbf{x}_i; \Theta)$$

Dataset: SMS Spam Classification

- Originially published in Almeida et al (2011)
- Download: http://www.dt.fee.unicamp.br/~tiago/smsspamcollection/
- Collection of 5,574 SMS messages with *spam* or *ham* classification
- Example:
  - SMS: *"Hi, the SEXYCHAT girls are waiting for you to text them. Text now for a great night chatting. send STOP to stop this service"*
  - Label: *spam*

Almeida et al (2011). "Contributions to the Study of SMS Spam Filtering: New Collection and Results". *J. Environ. Economics & Management*.

Text Representation: Bag-of-Words

- We will represent SMS messages as bag-of-words vectors
- We limit our vocabulary to the 300 most frequent words
- We represent each message by a 300-dimensional vector where each dimension counts the occurrence of a word
- The example SMS becomes:

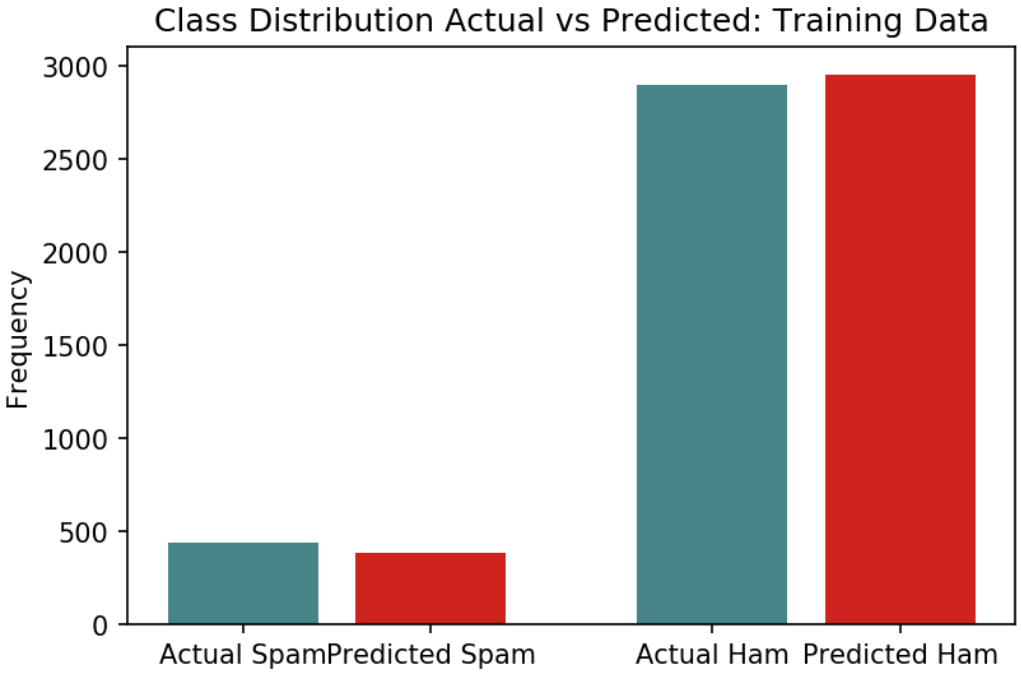
```
[0 2 1 1 1 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 2 0 0 0 0 0 0 0
 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

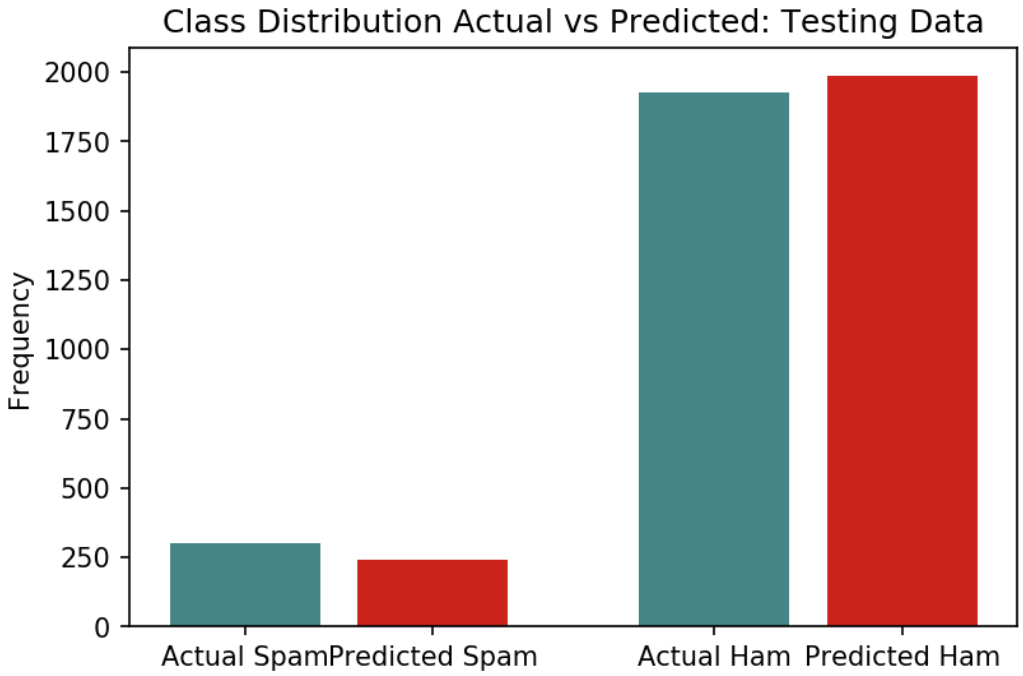
Ex 6: Linear Regression (Spam)

- Code: `06_linear_regression__spam.ipynb`
- For comparison, we will first try to tackle this dataset with normal linear regression and squared error loss
- Code identical to Ex 5 besides dataset

Ex 6: Results

	Precision	Recall	F1	Accuracy
Training	**0.98**	**0.86**	**0.91**	**0.98**
Testing	**0.96**	**0.77**	**0.86**	**0.97**





Ex 7: Logistic Regression (Spam)

- Code: `07_logistic_regression__spam.ipynb`
- Now with logistic regression
- Code identical to Ex 6, besides:

```
batch_x = tf.placeholder(tf.float32, shape=[None, num_features])
batch_y = tf.placeholder(tf.int32, shape=[None])

w = tf.Variable(tf.random_normal(shape=[num_features], mean=0, stddev=1))
b = tf.Variable(0.0)

logits = tf.tensordot(batch_x, w, 1) + b
y_prediction = tf.cast(tf.greater(logits, 0), tf.int32)

loss = tf.losses.sigmoid_cross_entropy(batch_y, logits)
# The above equivalent to the following but without numerical instability.
# loss = tf.reduce_mean(batch_y * -tf.log(tf.nn.sigmoid(logits))
#                       + (1 - batch_y) * -tf.log(1 - tf.nn.sigmoid(logits)))

train_op = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)
```

For discussion of numerical instability, see:
http://www.nowozin.net/sebastian/blog/streaming-log-sum-exp-computation.html

Ex 7: Logistic Regression (Spam)

- Code: `07_logistic_regression__spam.ipynb`
- Now with logistic regression
- Code identical to Ex 6, besides:

Let TensorFlow worry about implementing cross-entropy correctly

```
batch_x = tf.placeholder(tf.float32, shape=[None, num_features])
batch_y = tf.placeholder(tf.int32, shape=[None])

w = tf.Variable(tf.random_normal(shape=[num_features], mean=0, stddev=1))
b = tf.Variable(0.0)

logits = tf.tensordot(batch_x, w, 1) + b
y_prediction = tf.cast(tf.greater(logits, 0), tf.int32)

loss = tf.losses.sigmoid_cross_entropy(batch_y, logits)
# The above equivalent to the following but without numerical instability.
# loss = tf.reduce_mean(batch_y * -tf.log(tf.nn.sigmoid(logits))
#                       + (1 - batch_y) * -tf.log(1 - tf.nn.sigmoid(logits)))

train_op = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)
```

For discussion of numerical instability, see:
http://www.nowozin.net/sebastian/blog/streaming-log-sum-exp-computation.html

What are "logits"?

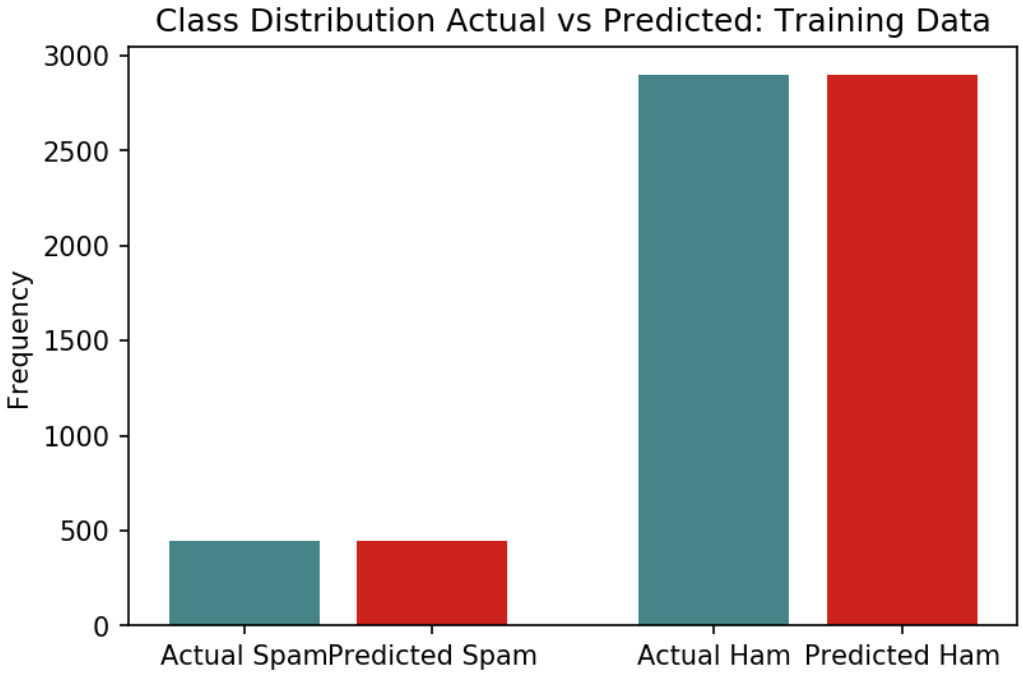
- The logit function is the inverse of the sigmoid function:

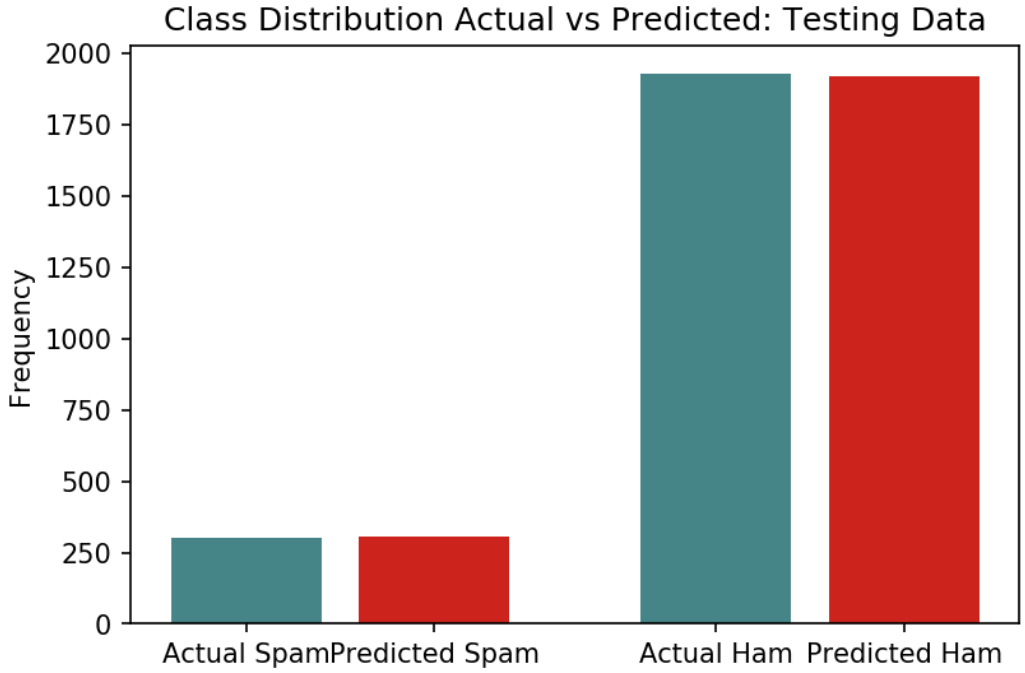
$$\text{logit}(x) = \sigma^{-1}(x) = -\log\left(\frac{1}{x} - 1\right)$$

- In machine learning any scores that are later turned into probabilities (via sigmoid or other functions) are commonly called **logits**

Ex 7: Results (compared to linear reg)

	Precision	Recall	F1	Accuracy
Training	**1.0** *(was **0.98**)*	**0.99** *(was **0.86**)*	**0.99** *(was **0.91**)*	**1.0** *(was **0.98**)*
Testing	**0.89** *(was **0.96**)*	**0.91** *(was **0.77**)*	**0.90** *(was **0.86**)*	**0.97** *(was **0.97**)*





Ex 7: Introspection

- One great thing about linear and logistic regression is that we have just one weight for each input feature
- Therefore, we can check the learned parameters for the importance of each input feature
- 50 most indicative words for spam from our classifier:

  50, txt, 18, chat, won, 150p, uk, min, service, reply, mins, claim, co, text, prize, b, around, com, other, special, msg, help, sms, mobile, s, message, live, send, call, urgent, an, free, go, hello, new, 2, r, been, over, you're, who, tone, days, cash, as, of, win, babe, www, x

Multinomial Logistic Regression

- Generalization of logistic regression
  - Input is still $\mathbf{x}_i = \left[x_{i1} \dots x_{id_{\text{in}}}\right] \in \mathbb{R}^{d_{\text{in}}}$
  - But arbitrarily many output classes: $\{y_1, \dots, y_{d_{\text{out}}}\}$
- Two parameters $\Theta = \{\mathbf{W}, \mathbf{b}\}$
  - weight matrix $\mathbf{W} = \begin{bmatrix} w_{11} & \dots & w_{d_{\text{in}}1} \\ \vdots & \ddots & \vdots \\ w_{1d_{\text{out}}} & \dots & w_{d_{\text{in}}d_{\text{out}}} \end{bmatrix}$
  - bias vector $\mathbf{b} = \left[b_1 \dots b_{d_{\text{out}}}\right]$
- Model function: $P(\hat{\mathbf{y}}_i = y_j \mid \mathbf{x}_i; \Theta) = \text{softmax}(\mathbf{x_i}\mathbf{W} + \mathbf{b})_j$

Softmax

- Model function: $P(\hat{\mathbf{y}}_i = y_j \mid \mathbf{x}_i; \Theta) = \text{softmax}(\mathbf{x_i W} + \mathbf{b})_j$
- The **softmax** function is applied to a vector of numbers in the range of $[-\infty, \infty]$ and turns them into a probability distribution, i.e. into the range $[0, 1]$ such that their sum is one:

$$\text{softmax}(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{d_{\text{out}}} e^{z_k}}$$

for a vector $\mathbf{z} = \left[z_1 \dots z_{d_{\text{out}}}\right]$ and $j = 1, \dots, d_{\text{out}}$

Dataset: Wine

- Originally published in Forina et al (1988)
- Download: https://archive.ics.uci.edu/ml/datasets/wine
- Chemical analysis of wines from three different cultivars
- Task: predict a wine's cultivar given 13 variables of analysis
  - *Alcohol*
  - *Malic acid*
  - *Ash*
  - ...

Forina et al (1988). "PARVUS: An Extendable Package of Programs for Data Exploration, Classification and Correlation". *Journal of Chemometrics*.

Ex 8: Multinomial Logistic Regression

» Code: `08_multinomial_logistic_regression__wine.ipynb`

» Similar to Ex 7, but with wine dataset and changed model:

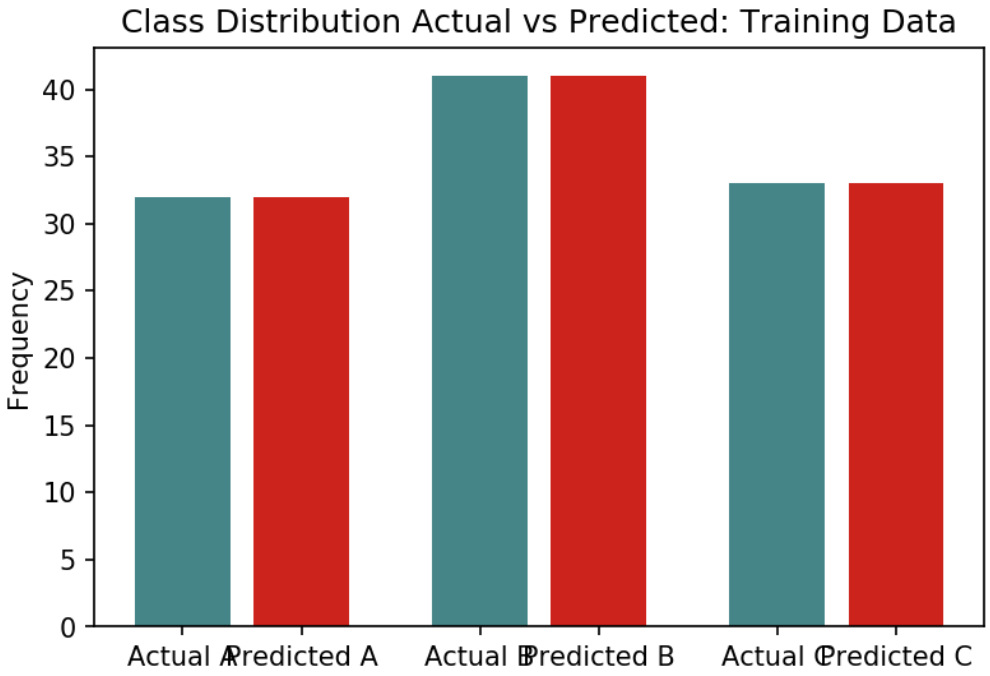
```
1  num_features = 13
2  num_classes = 3
3
4  # Model Definition
5  batch_x = tf.placeholder(tf.float32, shape=[None, num_features])
6  batch_y = tf.placeholder(tf.int32, shape=[None])
7
8  W = tf.Variable(tf.random_normal(shape=[num_features, num_classes],
9                                   mean=0, stddev=1))
10 b = tf.Variable(tf.zeros(shape=[num_classes]))
11
12 logits = tf.matmul(batch_x, W) + b
13 y_prediction = tf.argmax(logits, axis=-1, output_type=tf.int32)
14
15 loss = tf.losses.sparse_softmax_cross_entropy(batch_y, logits)
16 train_op = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)
```

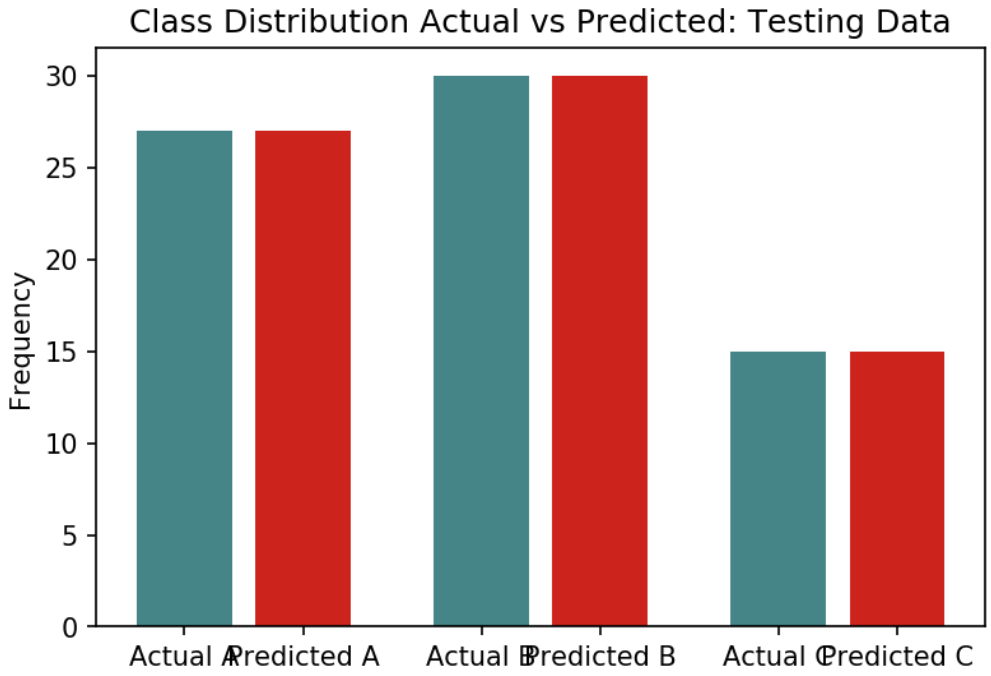
» Line 12 was `tf.tensordot(batch_x, w, 1) + b`

» Line 15 was `tf.losses.sigmoid_cross_entropy()`

Ex 8: Results

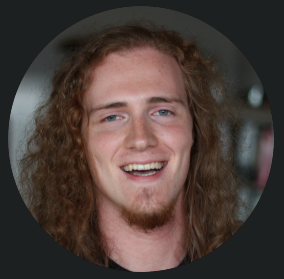
	Precision	Recall	F1	Accuracy
Training	**1.0**	**1.0**	**1.0**	**1.0**
Testing	**0.98**	**0.98**	**0.98**	**0.97**





Summary: Logistic Regression

- Logistic regression is the extension of linear regression to binary classifiaction
  - Model function $P(\hat{\mathbf{y}}_i = y_1 \mid \mathbf{x}_i; \Theta) = \sigma(\mathbf{x}_i \mathbf{w}^\intercal + b)$
- Multinomial regression is the extension to arbitrarily many output classes
  - Model function $P(\hat{\mathbf{y}}_i = y_j \mid \mathbf{x}_i; \Theta) = \text{softmax}(\mathbf{x_i W} + \mathbf{b})_j$
- Sigmoid function transforms from $[-\infty, \infty]$ to $[0, 1]$
- Softmax function takes vector of values and produces probability distribution
- Logits: scores that will be turned into probabilities



LUKASSCHMELZEISEN.

lukas@uni-koblenz.de
lschmelzeisen.com

24 Sep 2018

Introduction to Supervised Learning with TensorFlow

» Feedforward Neural Network

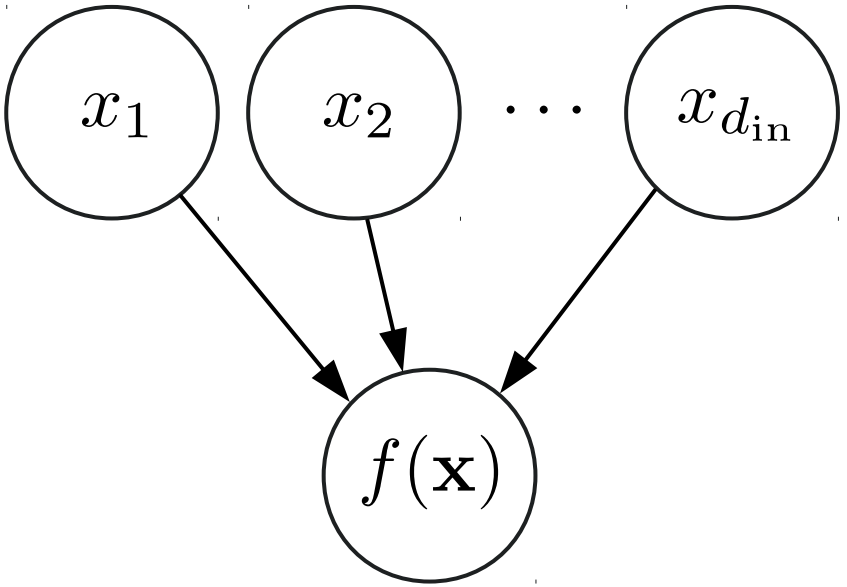
Artificial Neuron

- Generalization of linear and logistic regression

  $$f(\mathbf{x}) = \text{activation}(\mathbf{x}\mathbf{w}^\intercal + b)$$

  for input vector $\mathbf{x} = \begin{bmatrix} x_1 \dots x_{d_{\text{in}}} \end{bmatrix}$

- Parameters:
  - d_{in}-dimensional weight vector $\mathbf{w}$
  - bias scalar b
- Hyperparameter:
  - $\text{activation}(\cdot)$-function



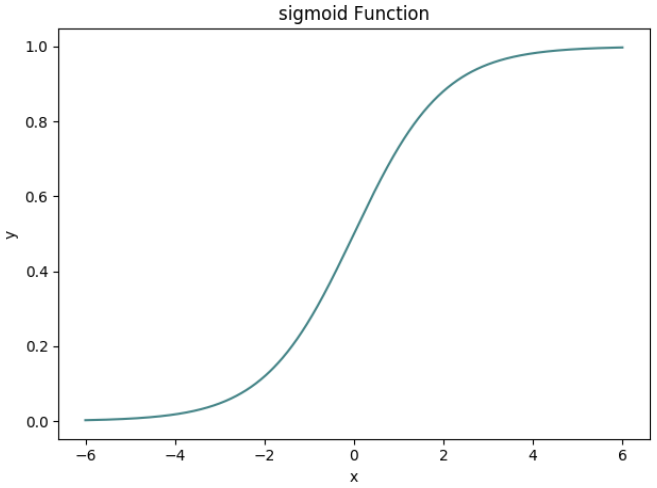
Activation Function

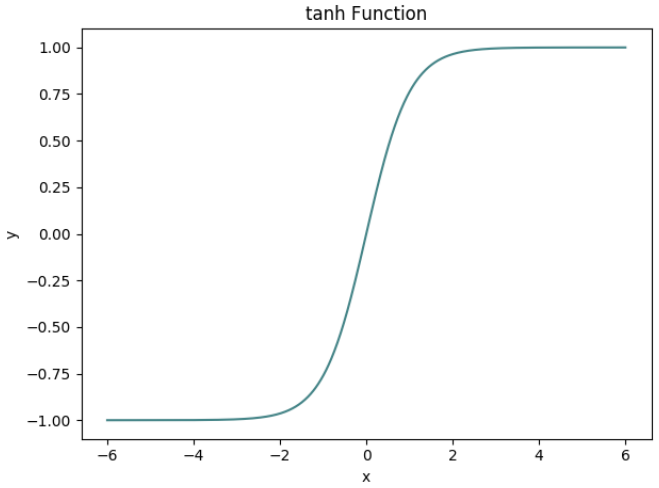
- Many different choices for activation function
- Must be non-linear
- Popular choices (needs to be evaluated for every problem):

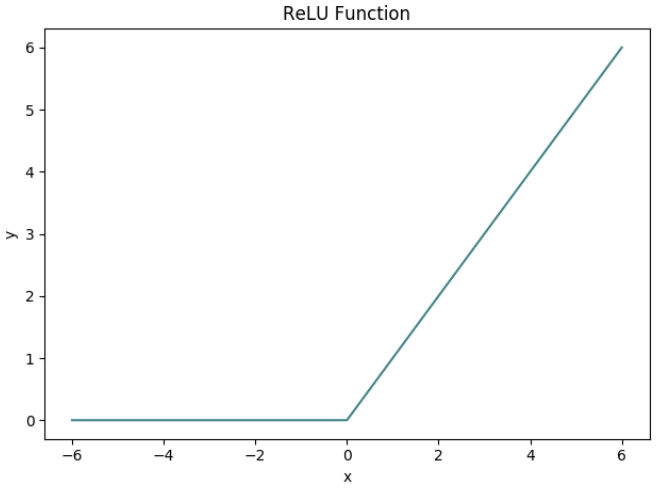
$$\text{sigmoid } \sigma(x) = \frac{1}{1+e^{-x}}$$

$$\tanh(x) = \frac{1-e^{-2x}}{1+e^{-2x}}$$

$$\text{ReLU}(x) = \max(0, x)$$

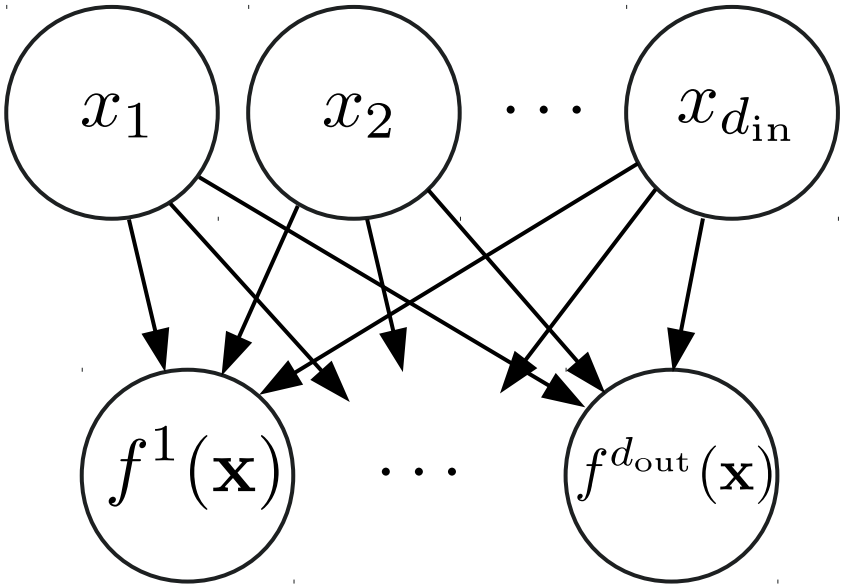






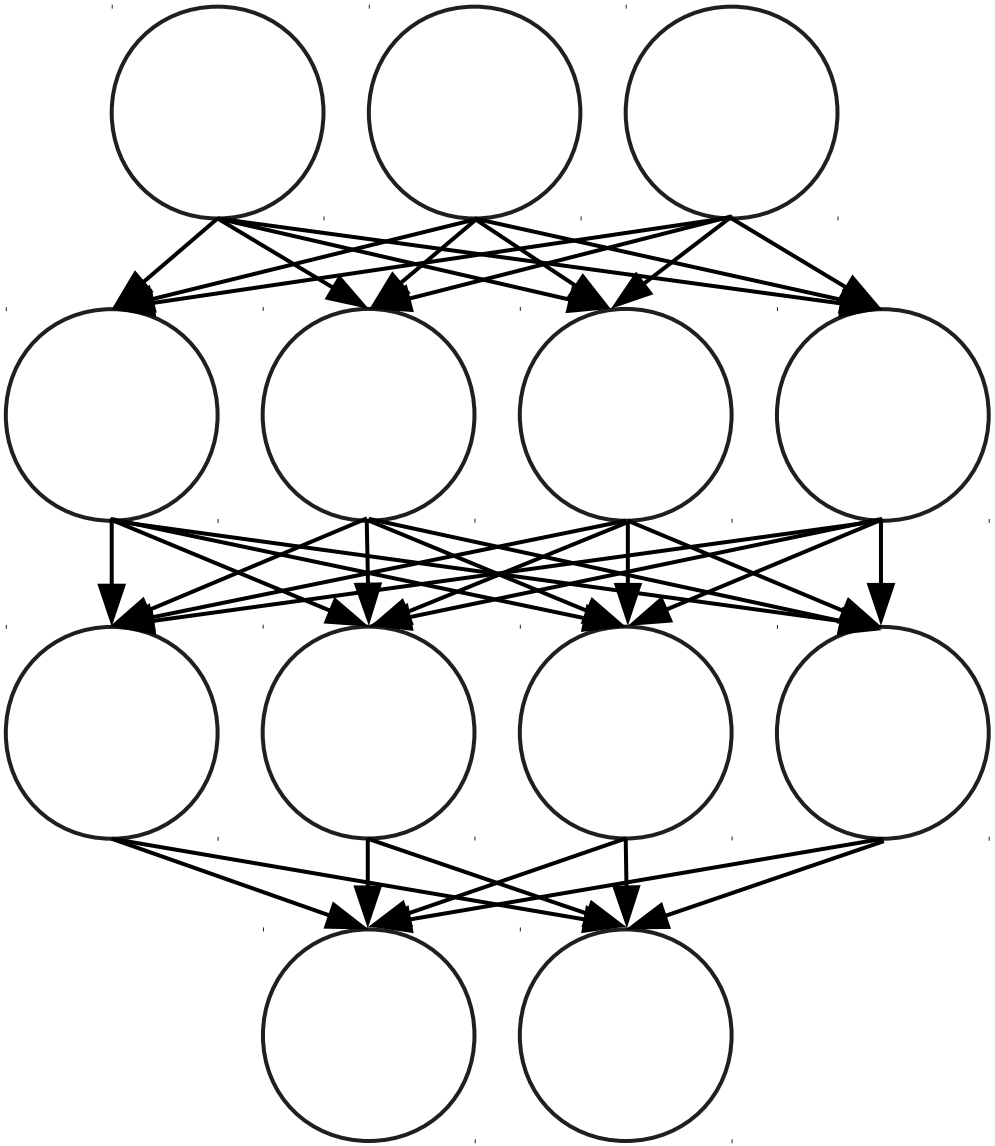
Dense Layer

- Grouping d_{out}-many neurons horizontally
- $\text{layer}(\mathbf{x}) = \text{activation}(\mathbf{x}\mathbf{W} + \mathbf{b})$
- Parameters:
  - $d_{\text{in}} \times d_{\text{out}}$ weight matrix $\mathbf{W}$
  - d_{out}-dimensional bias vector $\mathbf{b}$
- Activation function applied to every element of output vector

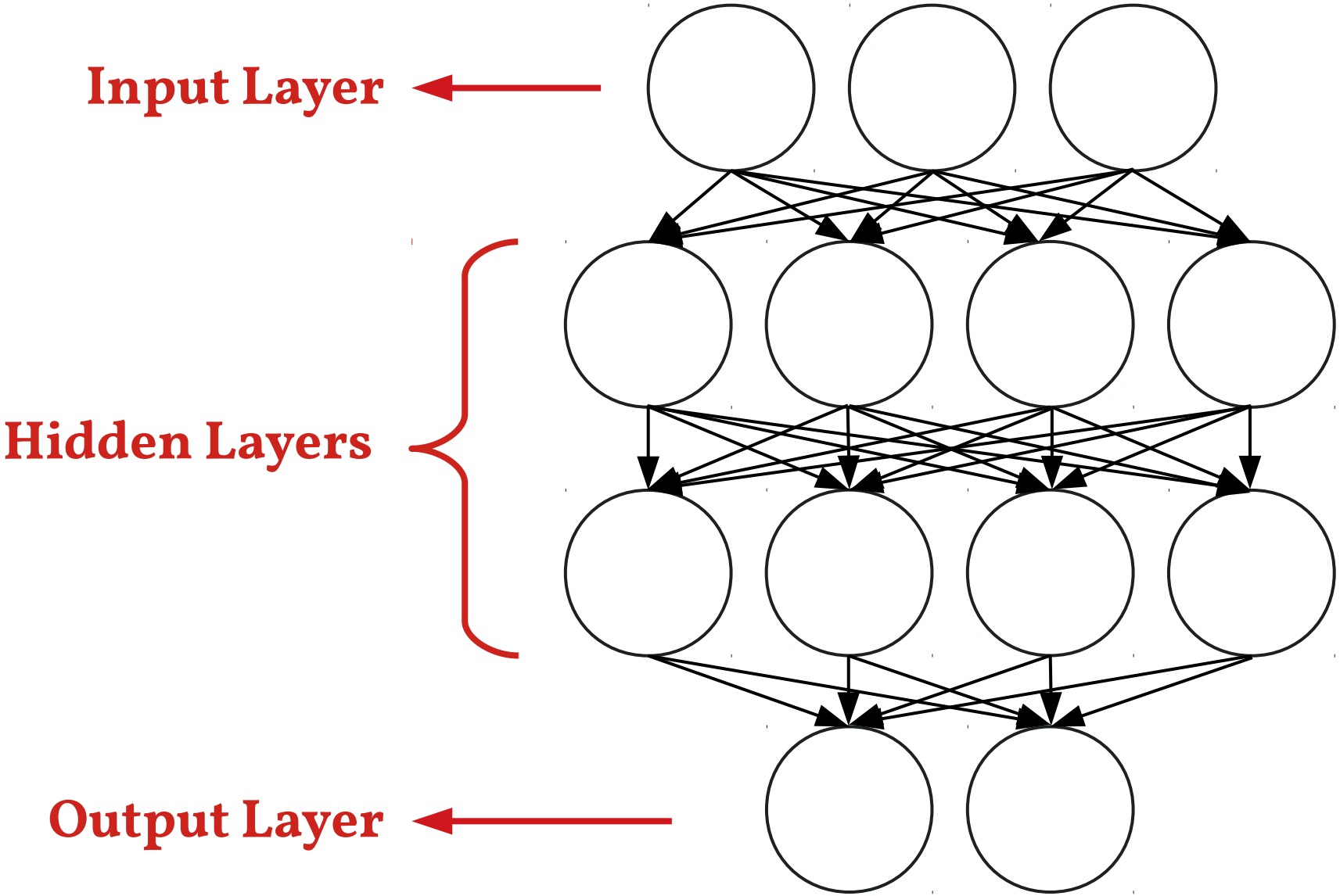


Feedforward Neural Network

- Also known as **multilayer perceptrons (MLP)**
- Stacking layers vertically
  - Number of inputs, layers, neurons per layer, and outputs are flexible
  - Non-input/-output layers are called **hidden layers**
- Better representation power
- For regression: no activation function in output layer
- For classification: softmax activation in output layer



Feedforward Neural Network



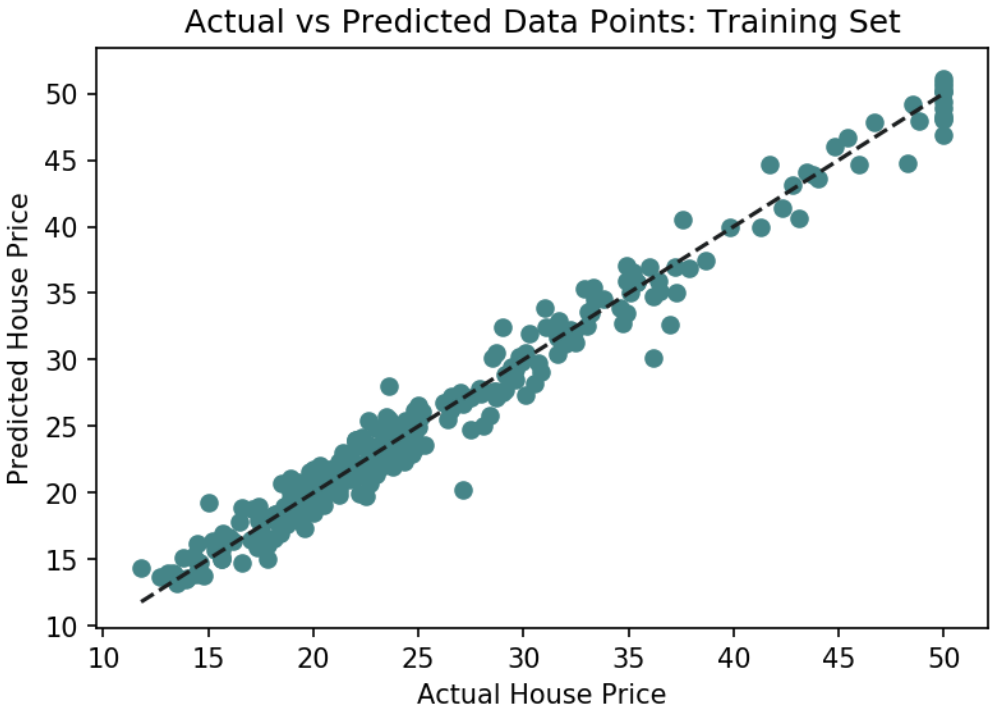
Ex 9: Feedforward Network (Housing)

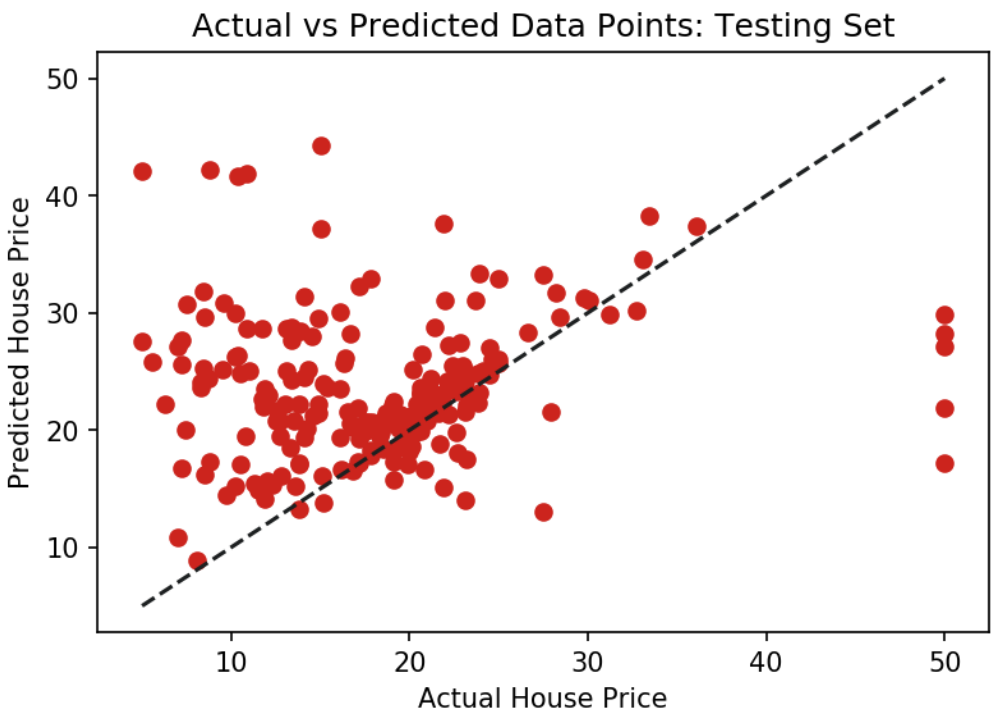
- Code: `09_feedforward_neural_network__housing.ipynb`
- Use feedforward neural network for regression on boston housing dataset, similar to Ex 5 (linear regression)

```
num_features = 13
num_hidden = 20

# Model Definition
batch_x = tf.placeholder(tf.float32, shape=[None, num_features])
batch_y = tf.placeholder(tf.float32, shape=[None, 1])

W1 = tf.Variable(tf.random_normal(shape=[num_features, num_hidden],
                                  mean=0, stddev=1))
b1 = tf.Variable(tf.zeros(shape=[num_hidden]))

W2 = tf.Variable(tf.random_normal(shape=[num_hidden, 1],
                                  mean=0, stddev=1))
b2 = tf.Variable(tf.zeros(shape=[1]))

h = tf.nn.sigmoid(tf.matmul(batch_x, W1) + b1)
y_prediction = tf.matmul(h, W2) + b2

loss = tf.reduce_mean((batch_y - y_prediction) ** 2)
train_op = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)
```

Ex 9: Results

- Mean squared error (lower is better)
  - Training data: **1.91** *(was **10.23** for linear regression)*
  - Testing data: **115.41** *(was **60.92** for linear regression)*

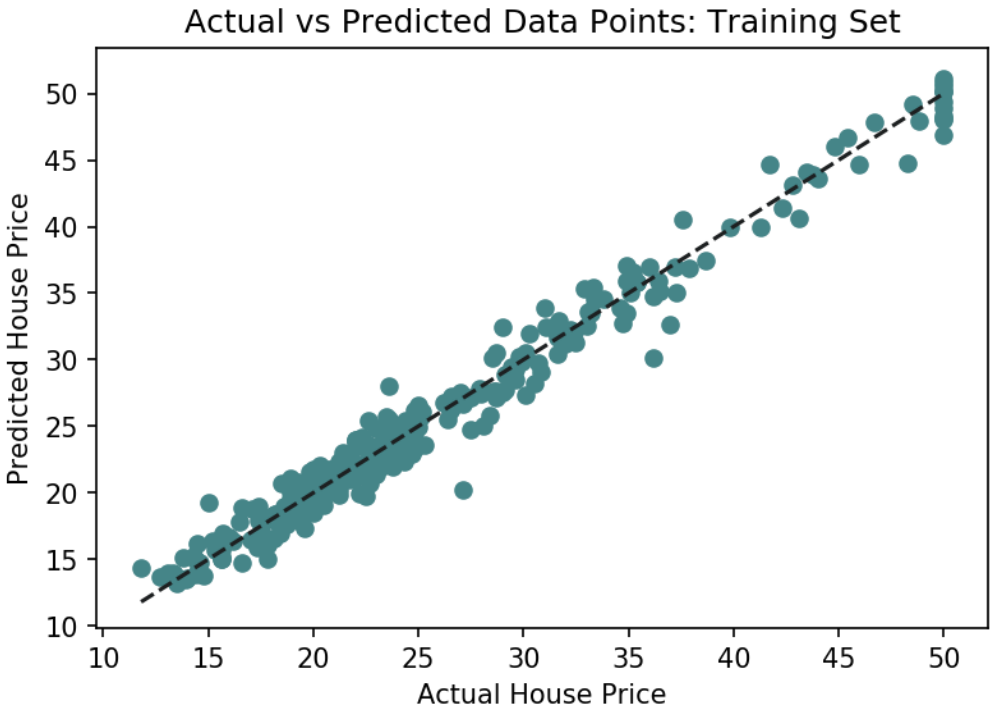


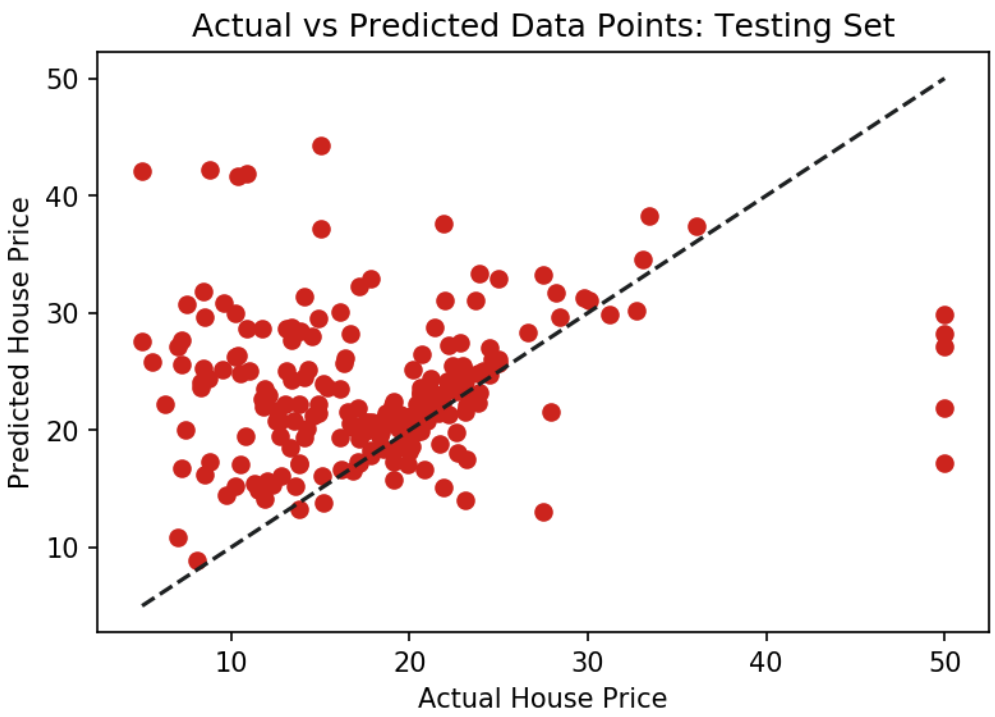


Ex 9: Results

Overfitting training data

- Mean squared error (lower is better)
  - Training data: **1.91** *(was **10.23** for linear regression)*
  - Testing data: **115.41** *(was **60.92** for linear regression)*





Ex 10: Using Advanced TensorFlow

- `10_feedforward_neural_network__housing__advanced.ipynb`
- We can use TensorFlow functions to improve some details
- Mostly same code as Ex 9

Ex 10: TensorFlow layers API

» Instead of:

```
W1 = tf.Variable(tf.random_normal(shape=[num_features, num_hidden],
                                  mean=0, stddev=1))
b1 = tf.Variable(tf.zeros(shape=[num_hidden]))

W2 = tf.Variable(tf.random_normal(shape=[num_hidden, 1],
                                  mean=0, stddev=1))
b2 = tf.Variable(tf.zeros(shape=[1]))

h = tf.nn.sigmoid(tf.matmul(batch_x, W1) + b1)
y_prediction = tf.matmul(h, W2) + b2
```

» we can do:

```
dense1 = tf.layers.Dense(num_hidden, activation=tf.nn.sigmoid)
dense2 = tf.layers.Dense(1)

h = dense1(batch_x)
y_prediction = dense2(h)
```

Ex 10: TensorFlow losses API

» Instead of:

```
1 loss = tf.reduce_mean((batch_y - y_prediction) ** 2)
```

» we can do:

```
1 loss = tf.losses.mean_squared_error(labels=batch_y, predictions=y_prediction)
```

(we already used this for `tf.losses.sigmoid_cross_entropy()`)

Ex 10: Adam Optimizer

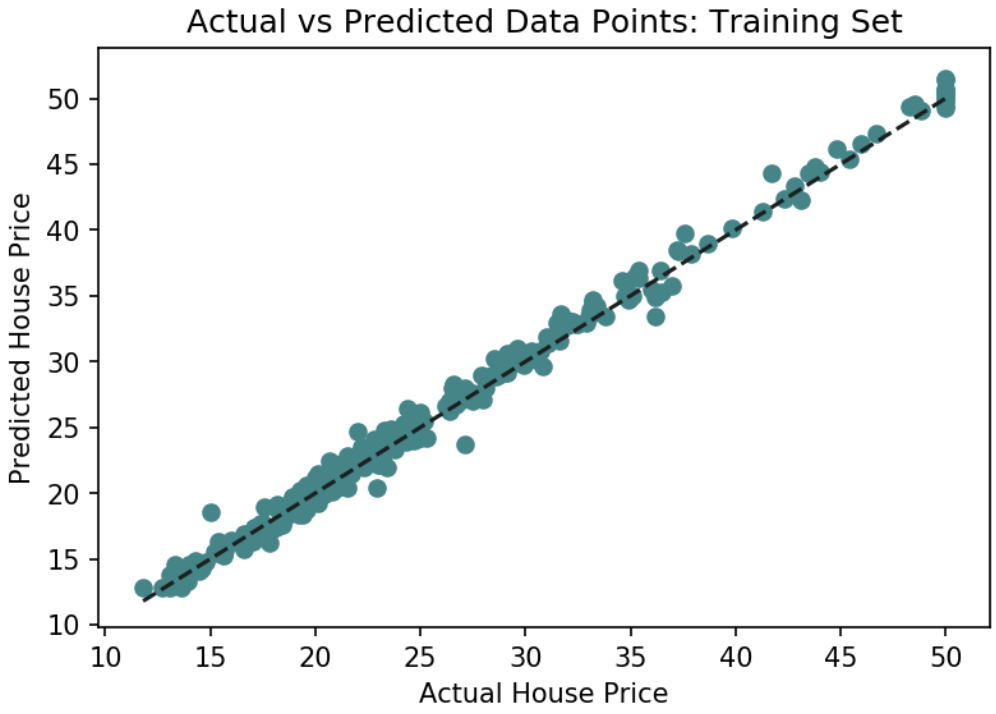
- Gradient descent is base algorithm
- Many common improvements, such as
  - Decreasing learning rate over time (per parameter)
  - Considering higher-order derivatives
- For a comprehensive overview of gradient descent variants, see: http://ruder.io/optimizing-gradient-descent/
- Adam (Kingma & Ba, 2014) is a good default choice for an optimizer:

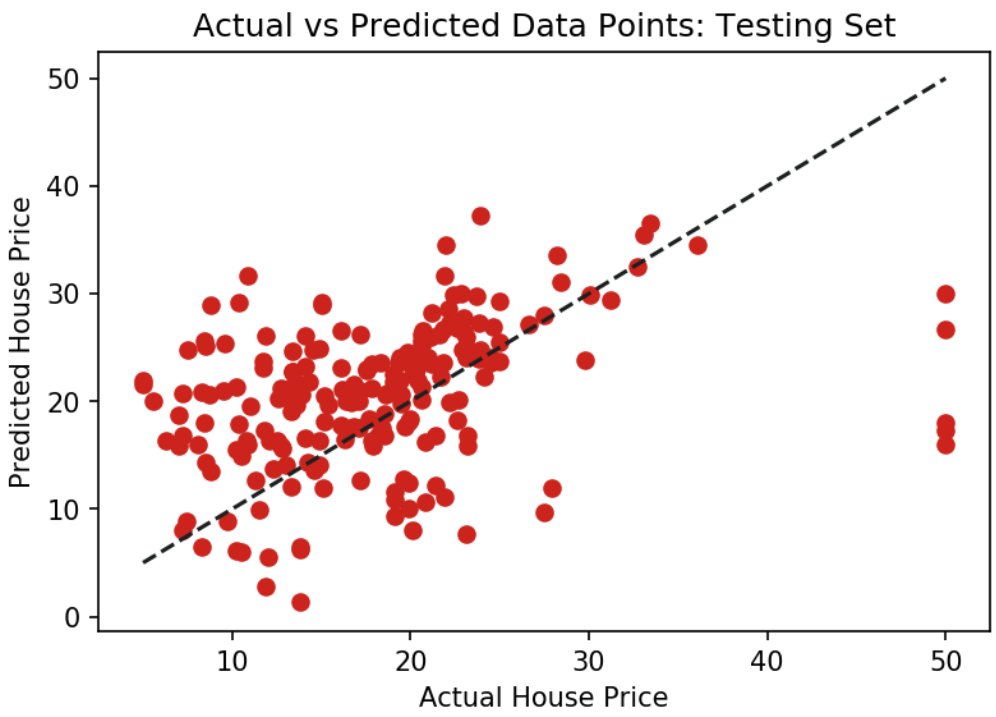
```
1 train_op = tf.train.AdamOptimizer(learning_rate).minimize(loss)
```

Kinga & Ba (2014). "Adam: A Method for Stochastic Optimization". *CoRR abs/1412.6980*.

Ex 10: Results

- Mean squared error (lower is better)
  - Training data: **0.68** *(was **1.91** for Ex 9, **10.23** for linear regression)*
  - Testing data: **74.95** *(was **115.41** for Ex 9, **60.92** for linear regression)*

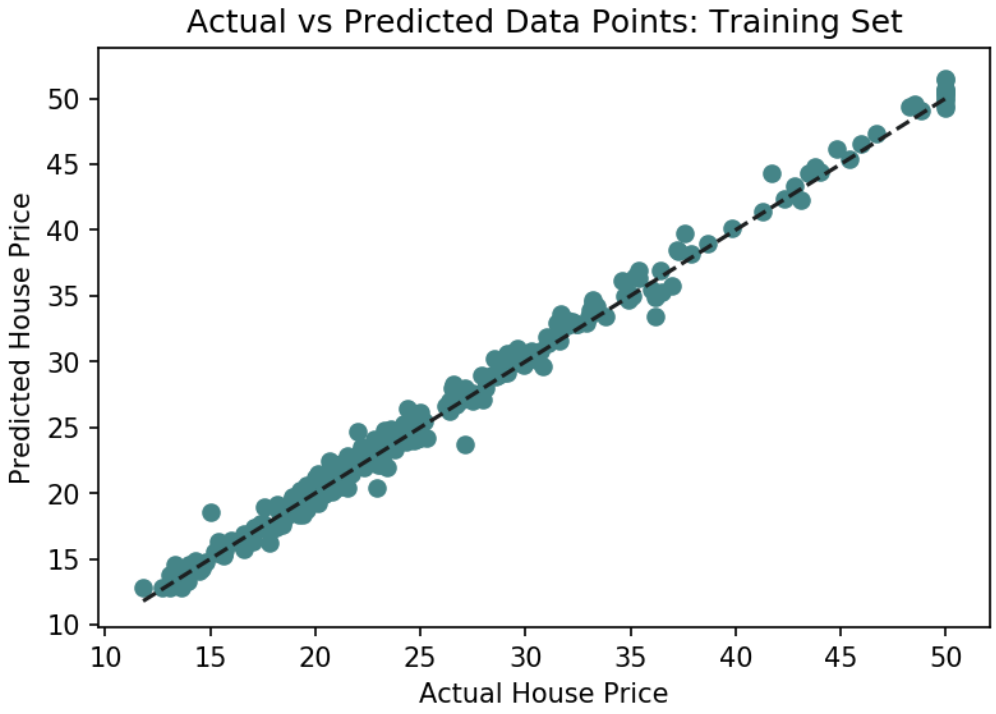


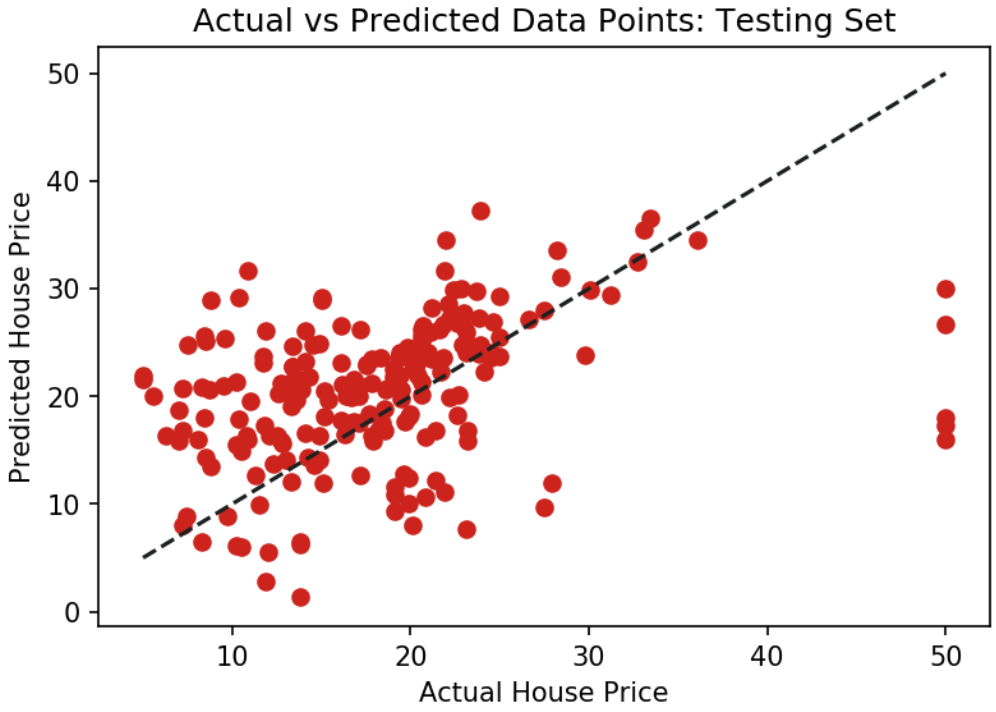


Ex 10: Results

still overfitting training data

- Mean squared error (lower is better)
  - Training data: **0.68** *(was **1.91** for Ex 9, **10.23** for linear regression)*
  - Testing data: **74.95** *(was **115.41** for Ex 9, **60.92** for linear regression)*





Regularization

- The current neural network model **overfits** the training data
  - It is able to fit the training data almost perfectly
  - Generalizes to unseen data worse than linear regression!
- One solution: **regularization**
  - Reduce model complexity for better generalization
  - Occam's razor: *"The simplest solution tends to be the right one"*
  - Implemented as modification to the loss function:

$$L_{\text{with regularization}}(\mathbf{y}_i, \hat{\mathbf{y}}_i) = L(\mathbf{y}_i, \hat{\mathbf{y}}_i) + L_{\text{regularziation term}}(\Theta)$$

    - $L(\mathbf{y}_i, \hat{\mathbf{y}}_i)$ is the "normal" part of the loss function we used until now
    - $L_{\text{regularziation term}}(\Theta)$ is a regularization term

L_2-Regularization

- Penalty term proportional to squared norms of weights:

  $$L_{\text{regularziation term}}(\Theta) = \lambda \|\Theta\|_2^2$$

  - λ is a hyperparameter that controls the regularization influence
  - $\|\mathbf{x}\|_2 = \sqrt{x_1^2 + \cdots + x_n^2}$ is the euclidean norm
- Usually only applied to feature weights and not biases
  - For $\Theta = \{\mathbf{W}_1, \mathbf{b}_1, \mathbf{W}_2, \mathbf{b}_2\}$ we would only do

    $$L_{\text{regularziation term}}(\Theta) = \lambda(\|\mathbf{W}_1\|_2^2 + \|\mathbf{W}_2\|_2^2)$$
- Weights close to zero have little impact on model complexity, while outlier weights can have a huge impact

Ex 11: Feedforward Network with L_2-Reg

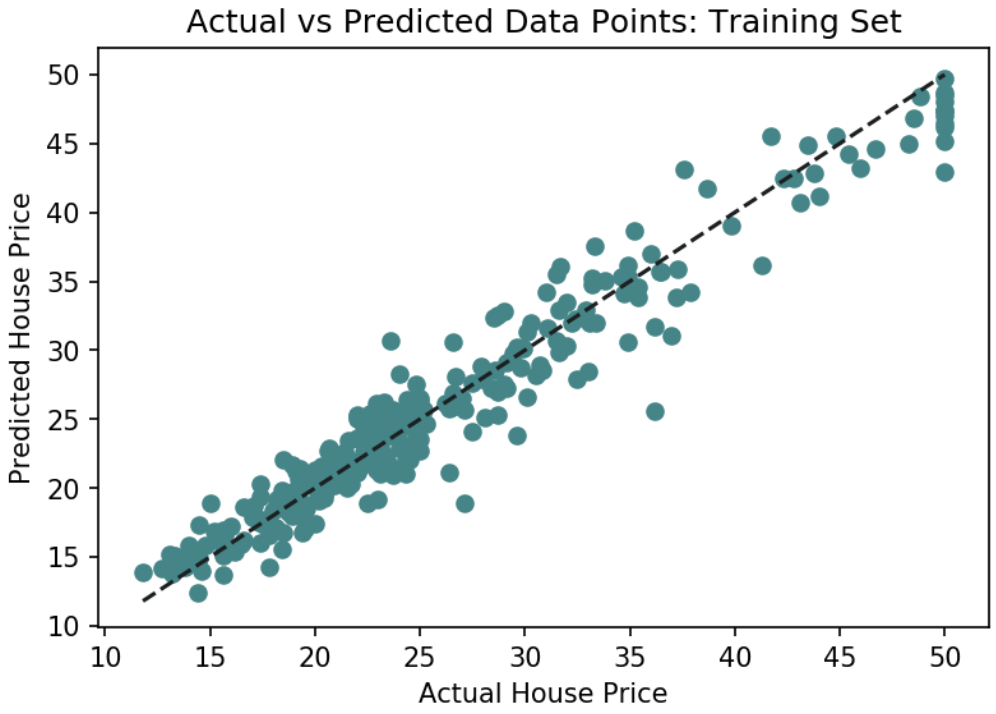
» `11_feedforward_neural_network__housing__with_regularization.ipynb`

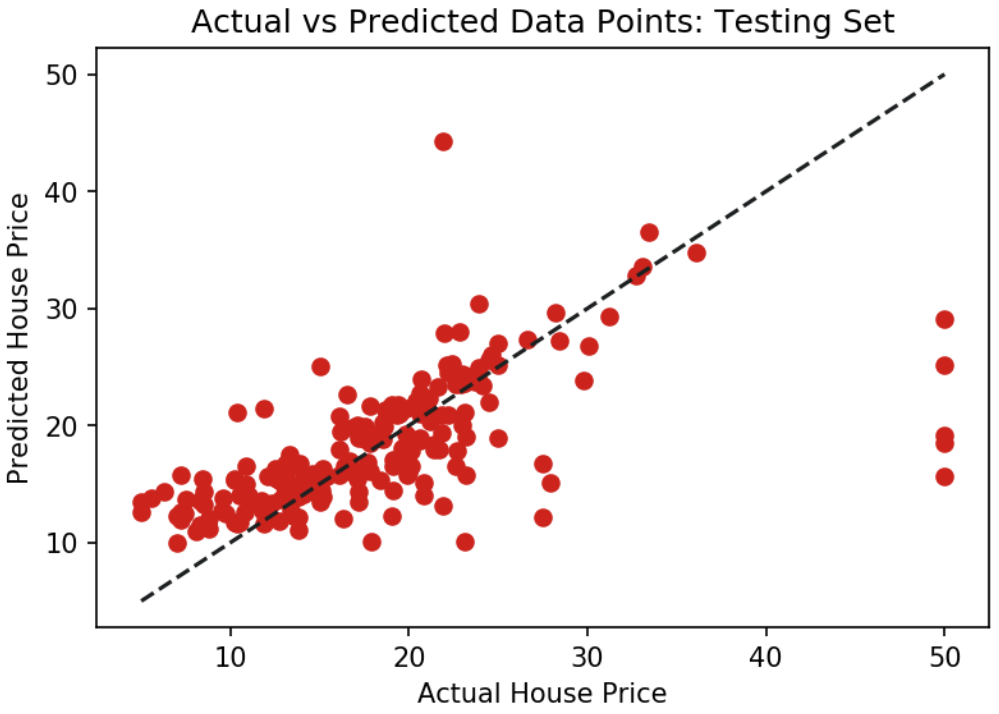
» Same code as Ex 10, but now with L_2-regularization

```
l2regularization_weight = 0.05

W1, b1 = dense1.weights
W2, b2 = dense2.weights

loss = (tf.losses.mean_squared_error(labels=batch_y, predictions=y_prediction)
        + l2regularization_weight * (tf.nn.l2_loss(W1) + tf.nn.l2_loss(W2)))
```

Ex 11: Results

- Mean squared error (lower is better)
  - Training data: **4.64** *(was* **0.68** *for Ex 10,* **1.91** *for Ex 9,* **10.23** *for LR)*
  - Testing data: **38.08** *(was* **74.95** *for Ex 10,* **115.41** *for Ex 9,* **60.92** *for LR)*

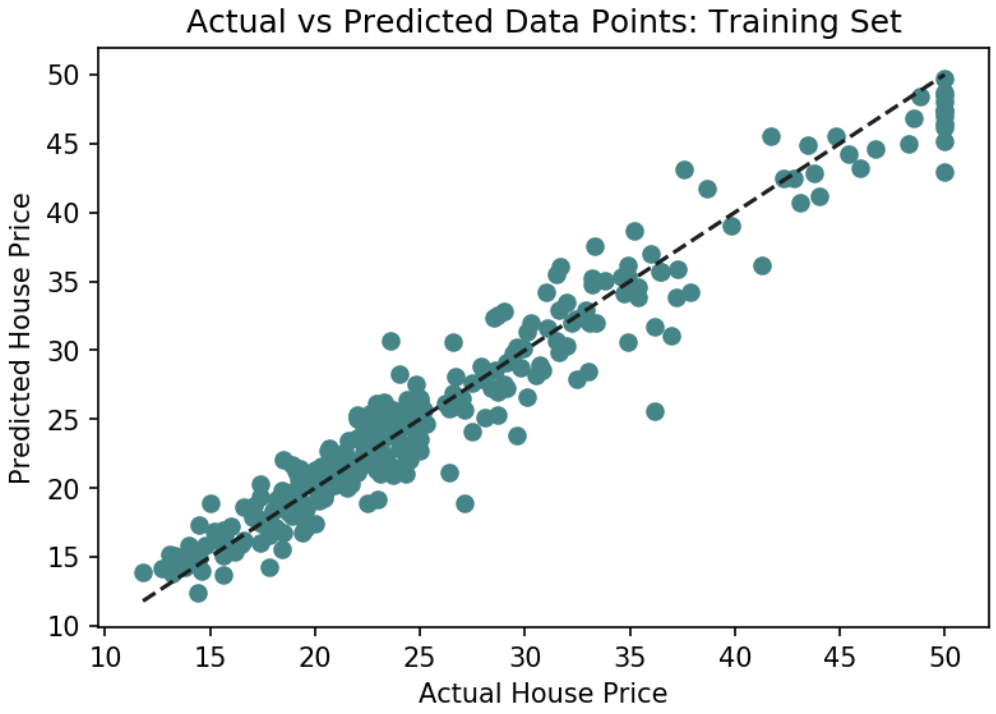


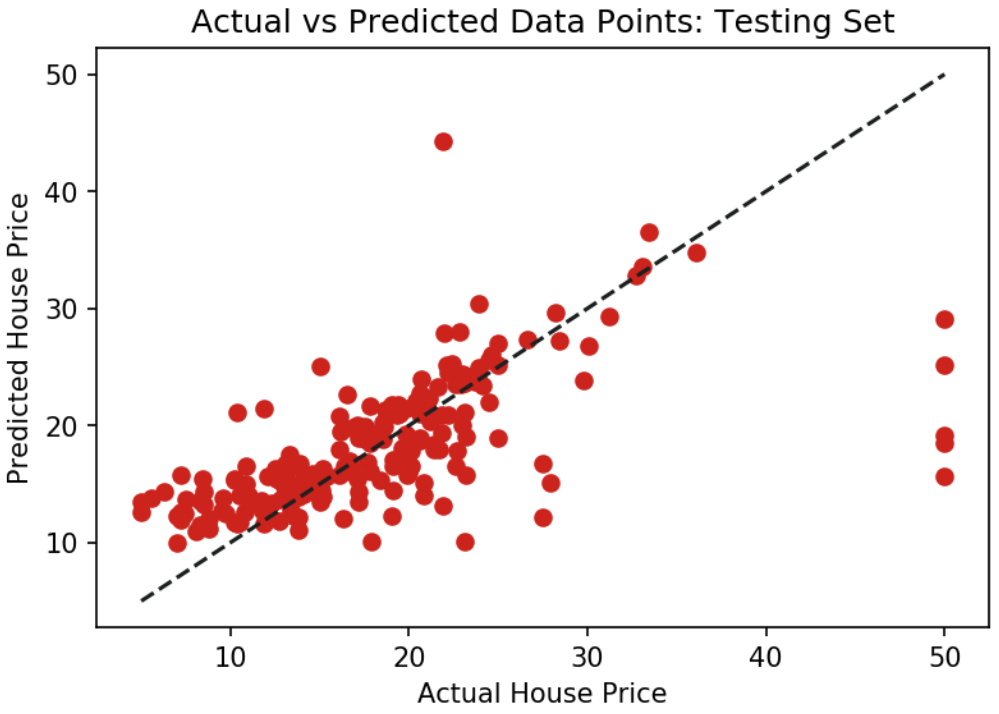


Ex 11: Results

trading worse training fit for better generalization

- Mean squared error (lower is better)
  - Training data: **4.64** *(was* **0.68** *for Ex 10,* **1.91** *for Ex 9,* **10.23** *for LR)*
  - Testing data: **38.08** *(was* **74.95** *for Ex 10,* **115.41** *for Ex 9,* **60.92** *for LR)*





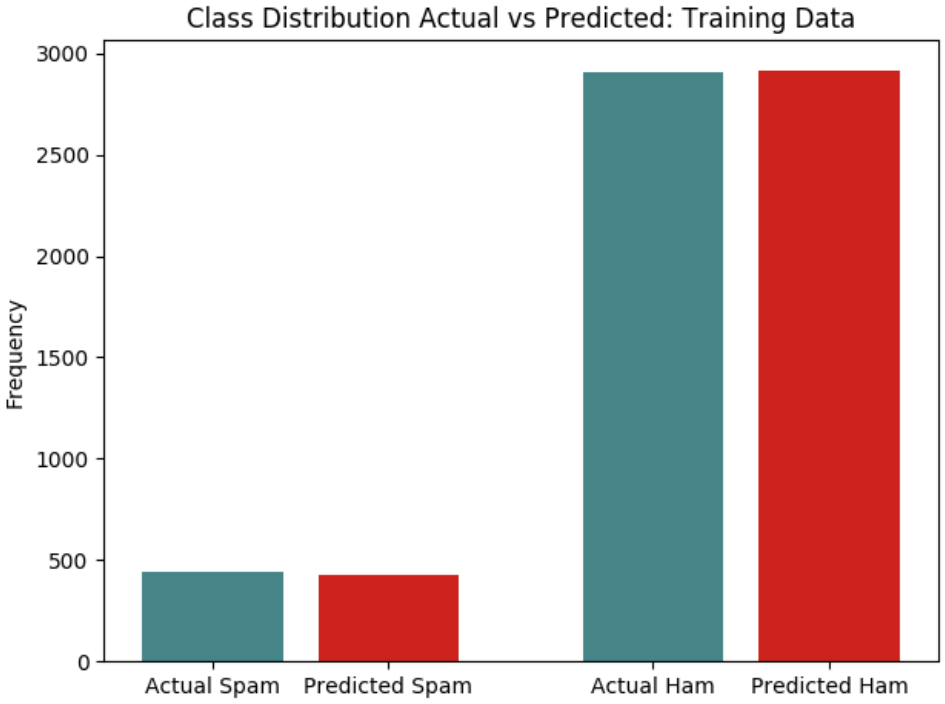
Ex 12: Feedforward Network (Spam)

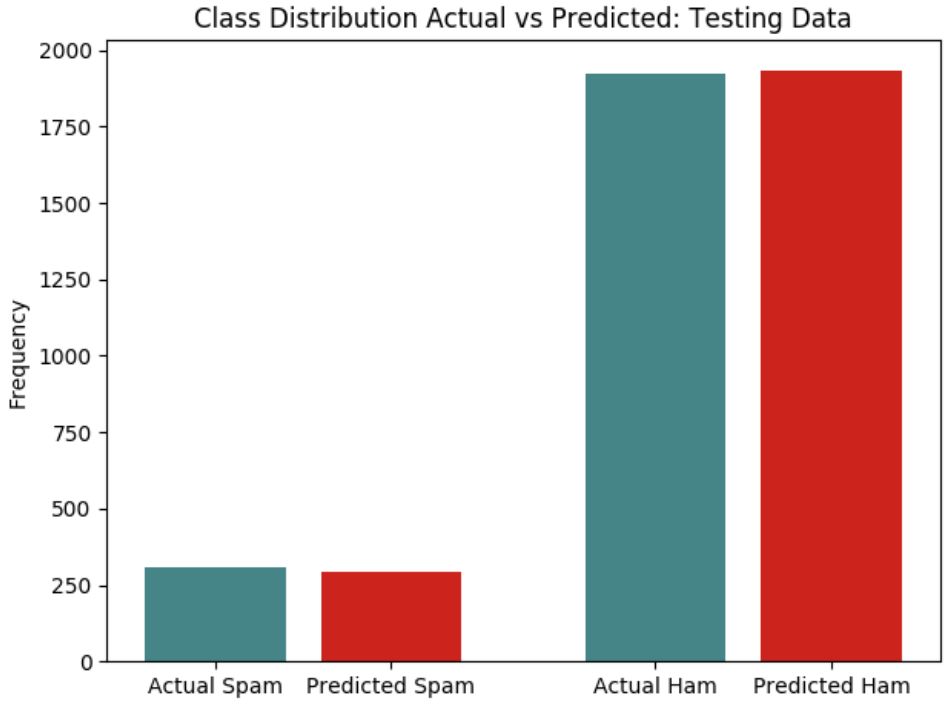
- Code: `12_feedforward_neural_network__spam.ipynb`
- Using neural network for classification (SMS spam)
- Code similar to Ex 11, but other dataset and cross entropy loss

```
batch_x = tf.placeholder(tf.float32, shape=[None, num_features])
batch_y = tf.placeholder(tf.int32, shape=[None])

dense1 = tf.layers.Dense(num_hidden, activation=tf.nn.sigmoid)
dense2 = tf.layers.Dense(num_hidden, activation=tf.nn.sigmoid)
dense3 = tf.layers.Dense(2)

h1 = dense1(batch_x)
h2 = dense2(h1)
logits = dense3(h2)
y_prediction = tf.argmax(logits, axis=-1, output_type=tf.int32)

W1, b1 = dense1.weights
W2, b2 = dense3.weights
W3, b3 = dense3.weights

loss = (tf.losses.sparse_softmax_cross_entropy(batch_y, logits)
        + l2regularization_weight * (
                tf.nn.l2_loss(W1) + tf.nn.l2_loss(W2) + tf.nn.l2_loss(W3)))

train_op = tf.train.AdamOptimizer(learning_rate).minimize(loss)
```

Ex 12: Results (compared to logistic reg)

	Precision	Recall	F1	Accuracy
Training	**1.0** *(was **1.0**)*	**0.99** *(was **0.99**)*	**1.0** *(was **0.99**)*	**1.0** *(was **1.0**)*
Testing	**0.92** *(was **0.89**)*	**0.91** *(was **0.91**)*	**0.92** *(was **0.90**)*	**0.98** *(was **0.97**)*



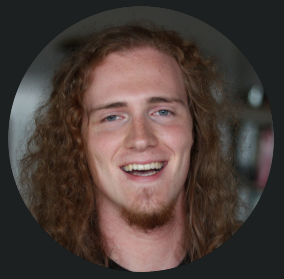


Summary: Feedforward Neural Networks

- Dense layer as horizontal grouping of artificial neurons

  $$\text{layer}(\mathbf{x}) = \text{activation}(\mathbf{x}\mathbf{W} + \mathbf{b})$$

- Activation functions need to be evaluated for every problem
- Neural networks are generalizations of linear and logistic regression and can do both regression and classifcation
- Neural networks are not always automatically better
- Regularization techniques reduce model complexity for better generalization to unseen data



LUKASSCHMELZEISEN.

lukas@uni-koblenz.de
lschmelzeisen.com

24 Sep 2018

Introduction to Supervised Learning with TensorFlow

» Conclusion

Not Mentioned: Hyperparameter Search

- Neural networks can include many hyperparameters
  - Learning rate, batch size, type of layers, activation function, number of layers, size of layers, regularization weight, …
- All hyperparameters can have a huge influence on model performance
- Finding a good model means manually trying out many different hyperparameter combinations

Not Mentioned: Other Layer Types

- **Convolutional Neural Networks (CNNs)** are used to detect local patterns
  - Convolution and pooling layers
- **Recurrent Neural Networks (RNNs)** are used to handly sequential input
  - Simple RNN, LSTM, or GRU layers
  - Bidirectional RNNs
- …

Not Mentioned: Advanced Techniques

- Dropout
- Batch Normalization
- Attention Mechanism
- Residual Connections
- ...

Not Mentioned: High-level TensorFlow

- This talk showcased TensorFlow's low-level APIs
  - Allows for better introduction to the field
  - Showcases how things work behind the scenes
- "Easier", more high-level APIs are available for many common scenarios

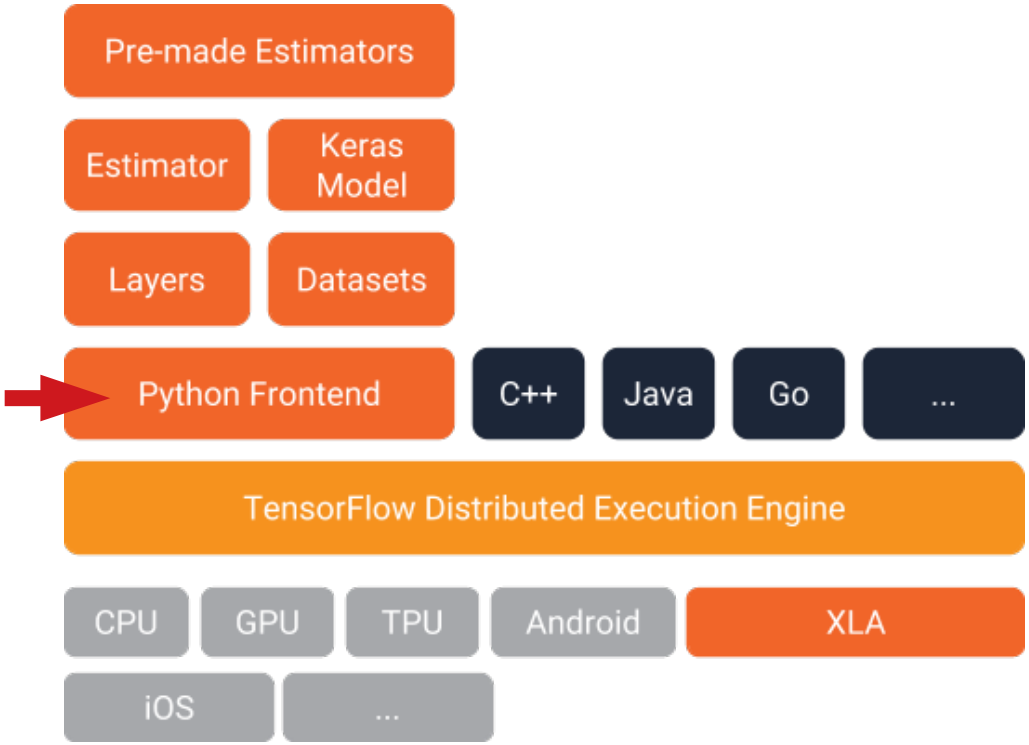


Image from "Introduction to TensorFlow Datasets and Estimators" by the TensorFlow Team
https://developers.googleblog.com/2017/09/introducing-tensorflow-datasets.html

Small Comparision of ML Frameworks

- Many competing open-source machine learning frameworks:
  - **TensorFlow (+ Keras)**
  - **PyTorch**
  - **Microsoft Cognitive Toolkit (CNTK)**
  - Apache MxNet
  - Chainer
  - DyNet
  - Deeplearning4j
  - …

TensorFlow

- Developed by Google
- Primary frontend language is Python, although bindings for many other languages exist (not always up-to-date)
- Largest community
- Supports scaling/deploying to production, as well as running on mobile
- Growing effort to provide pre-trained models
- Keras interface provides high-level abstraction layer suitable for beginners

Image from https://github.com/valohai/ml-logos

PyTorch

- Developed by Facebook
- Frontend language is Python
- Very large community
- Many pre-trained models available
- Said to have “cleaner” API than TensorFlow

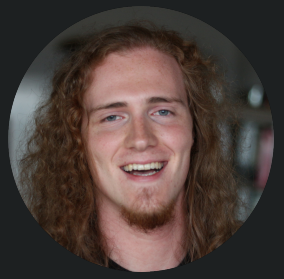
Images from https://github.com/pytorch/pytorch/blob/master/docs/source/_static/img/pytorch-logo-dark.svg

Microsoft Cognitive Toolkit (CNTK)

- Developed by Microsoft
- Full support for Python and C++
- Some pre-trained models available
- Relatively unused in research community

If You Want to Learn More…

- About **Deep Learning**:
  1. Google's Machine Learning Crash Course
     https://developers.google.com/machine-learning/crash-course/
  2. Andrew Ng's Deep Learning Specialization
     https://www.deeplearning.ai/courses/
  3. Goodfellow, Bengio, and Courville (2016). "Deep Learning". *MIT Press*.
     https://www.deeplearningbook.org/
- About **TensorFlow**:
  1. Official TensorFlow Tutorials
     https://www.tensorflow.org/tutorials/



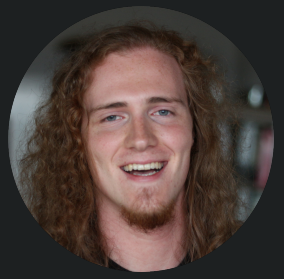
LUKASSCHMELZEISEN.

lukas@uni-koblenz.de
lschmelzeisen.com

24 Sep 2018

Summary

- Simple Linear Regression
  - Stochastic Gradient Descent
  - Computation Graph Abstraction
- Linear Regression
  - Tensors
  - Standardization
  - Mini-Batch Gradient Descent
- Logistic Regression
  - Sigmoid
  - Softmax
- Feedforward Neural Network
  - Dense Layers
  - Regularization



LUKASSCHMELZEISEN.

lukas@uni-koblenz.de
lschmelzeisen.com

24 Sep 2018

Questions?

- Simple Linear Regression
  - Stochastic Gradient Descent
  - Computation Graph Abstraction
- Linear Regression
  - Tensors
  - Standardization
  - Mini-Batch Gradient Descent
- Logistic Regression
  - Sigmoid
  - Softmax
- Feedforward Neural Network
  - Dense Layers
  - Regularization

Thank you for reading so far!

- Contact me if you have any questions
- Also, please forward all typos, errors, inaccuracies or inconsistencies you find in these slides or the code to me