Linear Regression

- *Regression* refers to a set of methods for modeling the relationship between one or more independent variables and a dependent variable.
- In the Natural and Social Sciences: characterizes the relationship between the inputs and outputs.
- In Machine Learning: concerned with *prediction* of a numerical (real) value.
- Common examples include predicting prices (of homes, stocks, etc.), predicting length of stay (for patients in the hospital), demand forecasting (for retail sales), among countless others.

Basic Elements of Linear Regression

- **Linear Regression**: simplest and most popular regression method.
- Assumes that the relationship between the independent variables $\mathbf{x}$ and the dependent variable y is linear $\rightarrow$ y can be expressed as a weighted sum of the elements in $\mathbf{x}$

Basic Elements of Linear Regression

- **Example**: Estimate the prices of houses (in pounds) based on their area (in square feet) and age (in years).

- Target (price) can be expressed as a weighted sum of the features (area and age):

$$\mathrm{price} = w_{\mathrm{area}} \cdot \mathrm{area} + w_{\mathrm{age}} \cdot \mathrm{age} + b.$$

- w_{area} and w_{age} are called *weights*, and b is called a *bias*
    - Weights determine the influence of each feature on our prediction and
    - Bias: the predicted price when all of the features take value 0.

Training Dataset

- To find the weights and bias of a model for predicting house prices, we need a training dataset consisting of sales for which we know the sale price, area, and age for each home.
    - n: the number of examples in our dataset. Each input: $\mathbf{x}^{(i)} = [x_1^{(i)}, x_2^{(i)}]^\top$ and the corresponding label: $y^{(i)}$.

Linear Model

- When inputs consist of d features, our prediction $\hat{y}$ is
$$\hat{y} = w_1 x_1 + \ldots + w_d x_d + b.$$
- Collecting all features into a vector $\mathbf{x} \in \mathbb{R}^d$ and all weights into a vector $\mathbf{w} \in \mathbb{R}^d$:
$$\hat{y} = \mathbf{w}^\top \mathbf{x} + b.$$
- Define *Design Matrix* $\mathbf{X} \in \mathbb{R}^{n \times d}$: contains one row for every example and one column for every feature. For a collection of features $\mathbf{X}$:
$$\hat{\mathbf{y}} = \mathbf{X}\mathbf{w} + b,$$
  - broadcasting is applied during the summation.

Finding the Weights from Training Data

- Given features of a training dataset $\mathbf{X}$ and corresponding (known) labels $\mathbf{y}$, the goal is to find $\mathbf{w}$ and b that given features of a new data point, its label will be predicted with the lowest error. We will need two more things:
    - a quality measure for some given model.
    - a procedure for updating the model to improve its quality.
- A noise term can be incorporated to account for measurement errors in the training dataset.

Loss Function

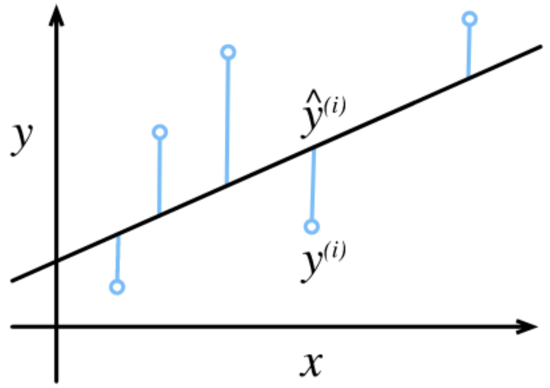
- The *loss function* quantifies the distance between the *real* and *predicted* value of the target.
- A non-negative number where smaller values are better and perfect predictions incur a loss of 0.

- Most popular loss function in regression is the squared error:

$$l^{(i)}(\mathbf{w}, b) = \frac{1}{2}\left(\hat{y}^{(i)} - y^{(i)}\right)^2.$$

  - $\hat{y}^{(i)}$: prediction, $y^{(i)}$: true label for example i
  - The error is only a function of the model parameters.

- **Example:** regression problem for a 1-d case:



Loss Function

- To measure the quality of a model on the entire dataset of n examples, we average (or sum) the losses on the training set:

$$L(\mathbf{w}, b) = \frac{1}{n} \sum_{i=1}^{n} l^{(i)}(\mathbf{w}, b) = \frac{1}{n} \sum_{i=1}^{n} \frac{1}{2}\left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right)^2.$$

- Training as an optimization problem: find parameters $(\mathbf{w}^*, b^*)$ that minimize the total loss across all training examples:

$$\mathbf{w}^*, b^* = \operatorname*{argmin}_{\mathbf{w}, b}\ L(\mathbf{w}, b).$$

Analytic Solution

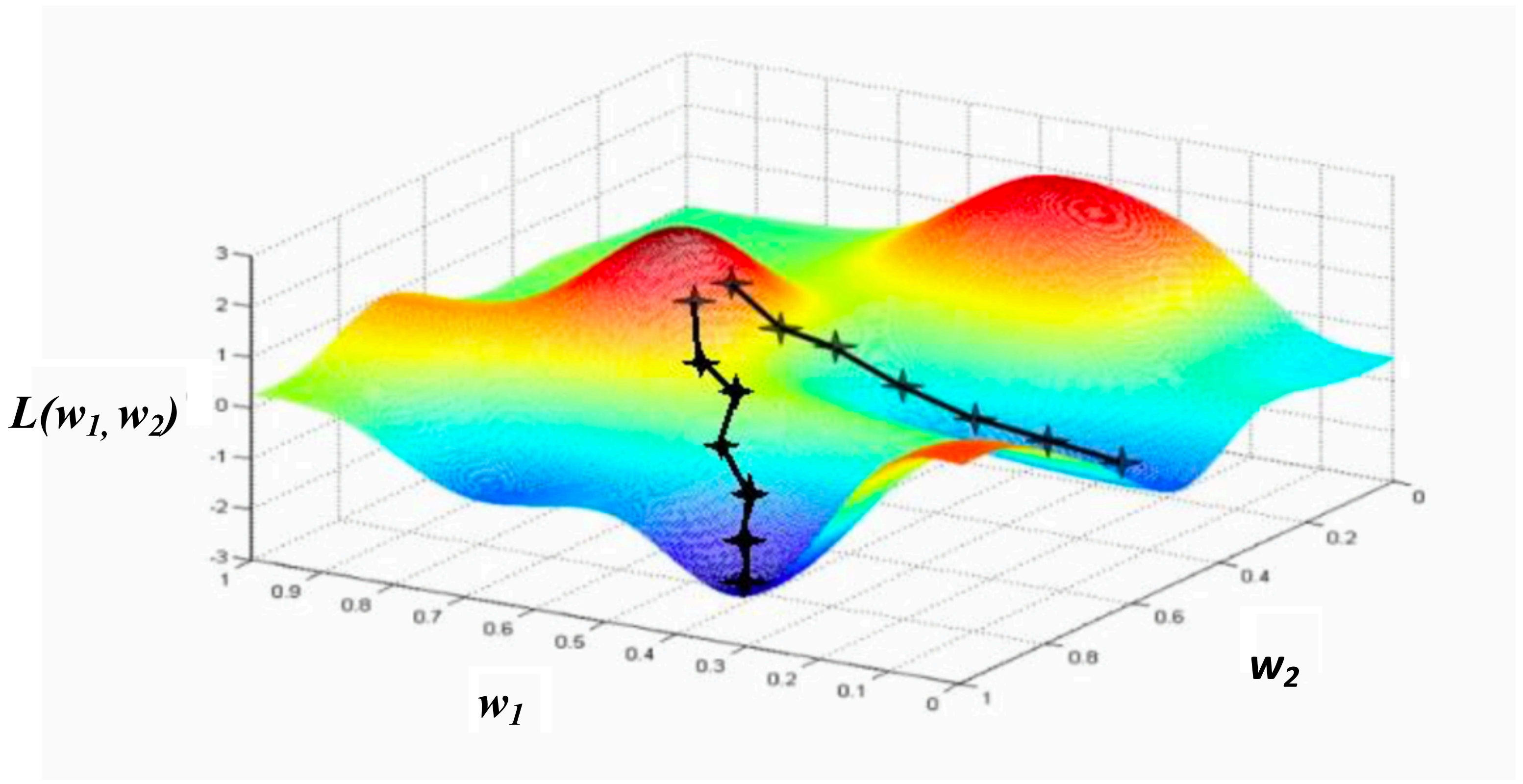
- Linear regression can be solved analytically by applying a simple formula.
- Our prediction problem is to minimize $\|\mathbf{y} - \mathbf{X}\mathbf{w}\|^2$.
    - Subsume bias b into the parameter $\mathbf{w}$ by appending a column to the design matrix consisting of all ones.
- Taking the derivative of the loss with respect to $\mathbf{w}$ and setting it equal to zero yields the analytic (closed-form) solution:

$$\mathbf{w}^* = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}.$$

- While simple problems like linear regression may admit analytic solutions, we should not get used to such good fortune :)

Gradient Descent (GD)

- Even in cases where we cannot solve the models analytically, it turns out that we can still train models effectively in practice.
- **Gradient Descent**: The key technique for optimizing nearly any deep learning model, consists of iteratively reducing the error by updating the parameters in the direction that incrementally lowers the loss function.



$L(w_1, w_2)$
w_1
w_2
3
2
1
0
-1
-2
-3
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
0.2
0.4
0.6
0.8
1

Updating the model weights using GD

- We can express the update mathematically as follows (∂ denotes the partial derivative):

$$\mathbf{w} \leftarrow \mathbf{w} - \frac{\eta}{N} \sum_{i=1}^{N} \nabla_{\mathbf{w}} l^{(i)}(\mathbf{w}, b),$$
$$b \leftarrow b - \frac{\eta}{N} \sum_{i=1}^{N} \frac{\partial l^{(i)}(\mathbf{w}, b)}{\partial b}.$$

- η is a positive scalar called the **Learning Rate**.

- We initialize the values of the model parameters, typically at random.

Reminder: Gradient vector

- We can concatenate partial derivatives of a multivariate function $f(\mathbf{x})$ with respect to all its input variables to obtain the *gradient* vector of the function.
- The gradient of the function $f(\mathbf{x})$ with respect to $\mathbf{x}$ is a vector of n partial derivatives:

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = \left[\frac{\partial f(\mathbf{x})}{\partial x_1}, \frac{\partial f(\mathbf{x})}{\partial x_2}, \ldots, \frac{\partial f(\mathbf{x})}{\partial x_n}\right]^\top,$$

where $\nabla_{\mathbf{x}} f(\mathbf{x})$ is often replaced by $\nabla f(\mathbf{x})$ when there is no ambiguity.

Minibatch Stochastic Gradient Descent

- GD takes the average of the derivative of the losses computed on every single example in the dataset.
    - this can be extremely slow: we must pass over the entire dataset before making a single update.

- **Minibatch Stochastic Gradient Descent:** Sample a random minibatch of examples every time to compute the update:

    1. In each iteration, randomly sample a minibatch $\mathcal{B}$ consisting of a fixed number of training examples.
    2. Compute the derivative (gradient) of the average loss on the minibatch with regard to the model parameters.
    3. Multiply the gradient by the learning rate η and subtract the resulting term from the current parameter values.

Minibatch Stochastic Gradient Descent

- The updates are given by:

$$\mathbf{w} \leftarrow \mathbf{w} - \frac{\eta}{|\mathcal{B}|} \sum_{i=1}^{|\mathcal{B}|} \nabla_{\mathbf{w}} l^{(i)}(\mathbf{w}, b),$$
$$b \leftarrow b - \frac{\eta}{|\mathcal{B}|} \sum_{i=1}^{|\mathcal{B}|} \frac{\partial l^{(i)}(\mathbf{w}, b)}{\partial b}.$$

- $|\mathcal{B}|$: the number of examples in each minibatch: the **Batchsize**

- For the case of quadratic loss and linear regression:

$$\nabla_{\mathbf{w}} l^{(i)}(\mathbf{w}, b) = \mathbf{x}^{(i)} \left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right) = \mathbf{x}^{(i)} \left(\hat{y}^{(i)} - y^{(i)}\right)$$

$$\frac{\partial l^{(i)}(\mathbf{w}, b)}{\partial b} = \left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right) = \left(\hat{y}^{(i)} - y^{(i)}\right)$$

- We observe that the gradient depends on the error between the prediction and the ground truth value: $\hat{y}^{(i)} - y^{(i)}$

Minibatch Stochastic Gradient Descent

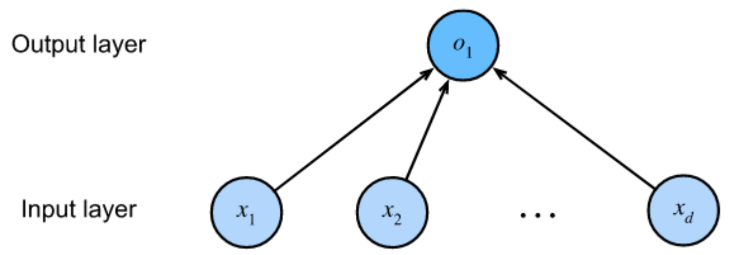
- Batchsize and Learning Rate are manually pre-specified
    - Called *Hyperparameters.*
    - Hyperparameter tuning is the process by which hyperparameters are chosen, based on results on a separate **Validation Set**.
- After training for some predetermined number of iterations (or until some other stopping criteria are met), we record the estimated model parameters, denoted $\hat{\mathbf{w}}, \hat{b}$.
- These parameters will not be the exact minimizers of the loss because, although the algorithm converges slowly towards the minimizers it cannot achieve it exactly in a finite number of steps.

Making Predictions with the Learned Model

- Given the learned linear regression model $\hat{\mathbf{w}}^\top \mathbf{x} + \hat{b}$, we can now estimate the price of a new house (not contained in the training data) given its area x_1 and age x_2.
  - Commonly called **Prediction** or **Inference**.

From Linear Regression to Deep Networks

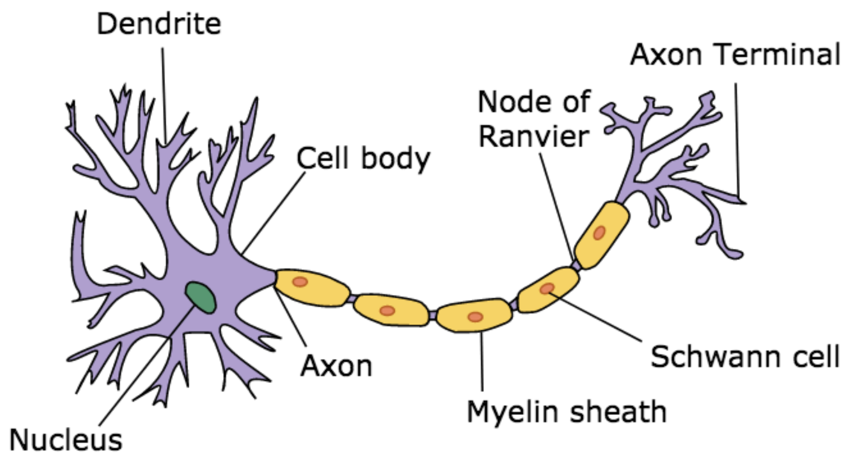
- A visualization of a linear regression model as a neural network:



- The above diagram highlights the connectivity pattern i.e. how each input is connected to the output
  - does not show the weights or biases.
- Inputs are $x_1, \dots, x_d$, so the *feature dimensionality* in the input layer is d.
- The output of the network is o_1, so the *number of outputs* in the output layer is 1.
- The *number of layers* for the neural network above is 1.
- We can think of linear regression models as neural networks consisting of just a single artificial neuron, or as single-layer neural networks.
- Since every input is connected to every output (in this case there is only one output), we can regard this as a *fully-connected layer* or *dense layer*.

Biology

- A biological neuron consists of *dendrites* (input), the *nucleus* (CPU), the *axon* (output wire), and the *axon terminals* (output), connecting to other neurons via *synapses*.



- Information x_i arriving from other neurons (or environmental sensors such as the retina) is received in the dendrites.
- x_i is weighted by *synaptic weights* w_i determining the effect of the inputs (e.g., activation or inhibition via the product $x_i w_i$).
- The weighted inputs arriving from multiple sources are aggregated in the nucleus as a weighted sum $y = \sum_i x_i w_i + b$, and this information is then sent for further processing in the axon y, typically after some nonlinear processing via $\sigma(y)$.
- From there it either reaches its destination (e.g., a muscle) or is fed into another neuron via its dendrites.

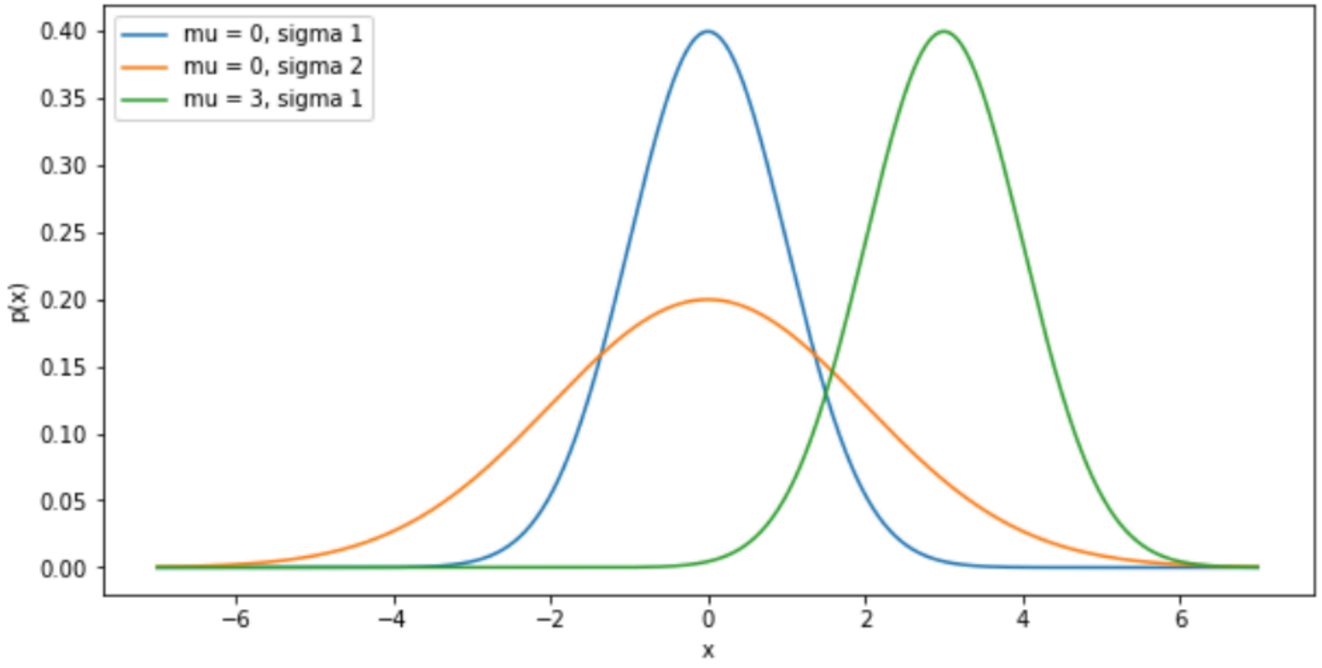
The Normal Distribution and Squared Loss

- Linear regression was invented by Gauss in 1795, who also discovered the normal distribution (also called the *Gaussian*).
- It turns out that there is connection between the normal distribution and linear regression
- The probability density of a normal distribution with mean μ and variance σ^2 (standard deviation σ) is given as:
$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(x - \mu)^2\right).$$
- Below we define a Python function to compute the normal distribution.

```
In [6]: def normal(x, mu, sigma):
            p = 1 / math.sqrt(2 * math.pi * sigma**2)
            return p * torch.exp(-0.5 / sigma**2 * (x - mu)**2)
```

We can now visualize various normal distributions.

In [7]:
```python
# Input values
x = torch.arange(-7, 7, 0.01)
# Mean and standard deviation pairs
params = [(0, 1), (0, 2), (3, 1)]
a = [normal(x, mu, sigma) for mu, sigma in params]
[plt.plot(x, a[i], label="mu = {}, sigma {}".format(params[i][0], params[i][1])) f
or i in range(len(a))]
plt.legend(loc="upper left");
plt.xlabel('x');
plt.ylabel('p(x)');
fig = plt.gcf()
fig.set_size_inches(10, 5, forward=True)
plt.show()
```



Summary

- Key ingredients in a machine learning model are training data, a loss function, an optimization algorithm, and quite obviously, the model itself.
- Vectorizing makes everything better (mostly math) and faster (mostly code).
- Minimizing an objective function and performing maximum likelihood estimation can mean the same thing.
- Linear regression models are neural networks, too.