CS6421: Deep Neural Networks

Gregory Provan

Spring 2020

Lecture 15: Gradient Descent and Learning Rates

Overview

- Practical aspects of gradient descent
  - algorithm
  - Inference steps
- Learning rate
- Regression example

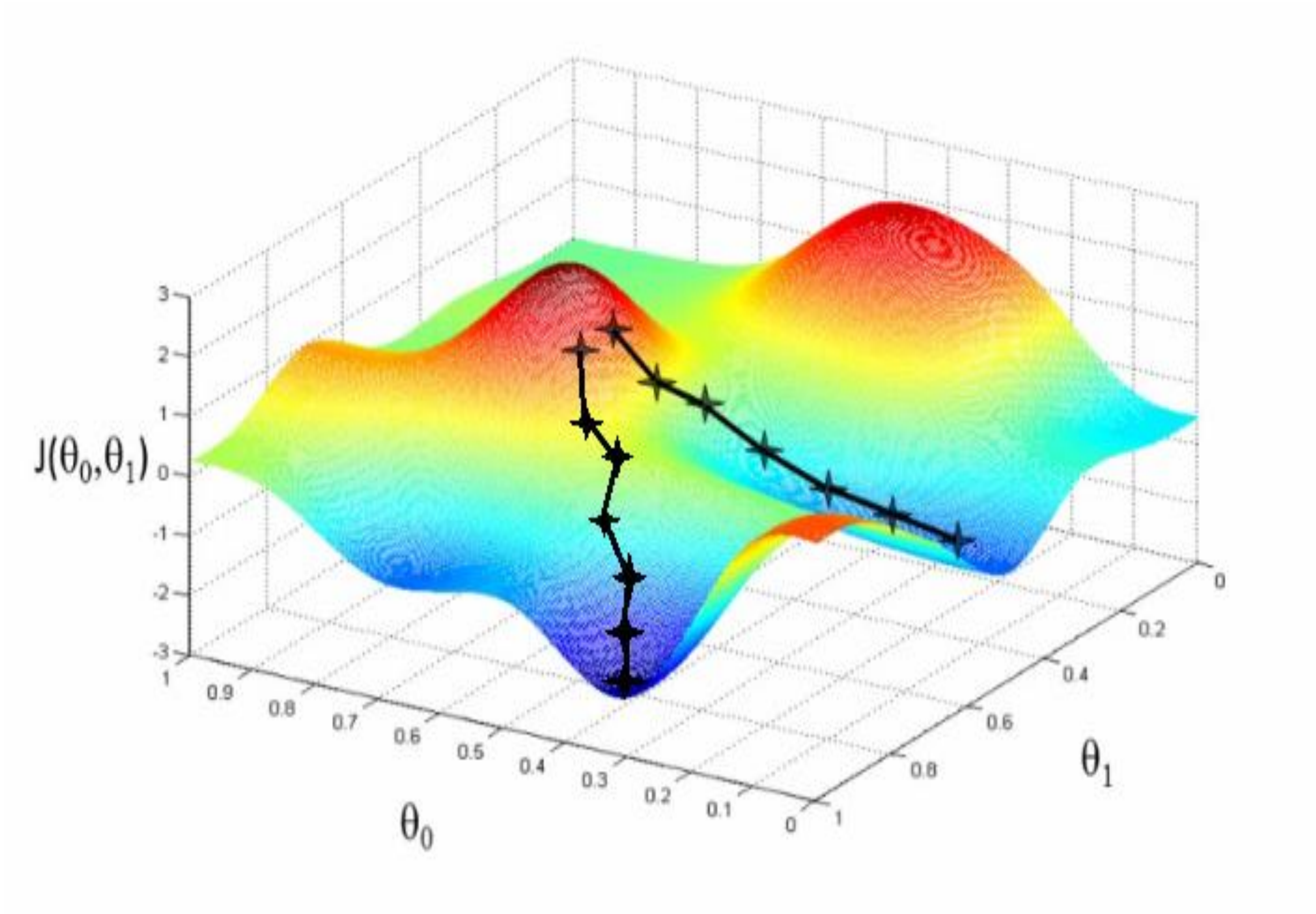
GRADIENT DESCENT ALGORITHM

- At a theoretical level, gradient descent is an algorithm that minimizes functions. Given a function defined by a set of parameters, gradient descent starts with an initial set of parameter values and iteratively moves toward a set of parameter values that minimize the function. This iterative minimization is achieved using calculus, taking steps in the negative direction of the function gradient.
- Minimize cost function J
- Gradient descent
  - Used all over machine learning for minimization
- Start by looking at a general J() function
- Problem
  - We have $J(\theta_0, \theta_1)$
  - We want to get **min $J(\theta_0, \theta_1)$**
- Gradient descent applies to more general functions
  - $J(\theta_0, \theta_1, \theta_2 \dots \theta_p)$
  - min $J(\theta_0, \theta_1, \theta_2 \dots \theta_p)$

How does it work?

- Start with initial guesses
  - Start at 0,0 (or any other value)
  - Keeping changing θ_0 and θ_1 a little bit to try and reduce $J(\theta_0,\theta_1)$
- Each time you change the parameters, you select the gradient which reduces $J(\theta_0,\theta_1)$ the most possible
- Repeat
- Do so until you converge to a local minimum
- Has an interesting property
  - Where you start can determine which minimum you end up

Visualisation



Do the following until convergence

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1) \text{ for } j = 0 \text{ or } 1$$

- What does this all mean?
  - Update θ_j by setting it to ($\theta_j - \alpha$) times the partial derivative of the cost function with respect to θ_j
  - α (alpha)
    - is a number called the **learning rate**
    - Controls how big a step you take
      - If α is big have an aggressive gradient descent
      - If α is small take tiny steps
- There is a slightly about how this gradient descent algorithm is implemented
  - Do this for θ_0 and θ_1
  - For j = 0 and j = 1 means we **simultaneously** update both
  - How do we do this?
    - Compute the right hand side for both θ_0 and θ_1
      - So we need a temp value
    - Then, update θ_0 and θ_1 at the same time

$$temp0 = \theta_0 - \alpha \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1)$$

$$temp1 = \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1)$$

$$\theta_0 = temp0$$

$$\theta_1 = temp1$$

STEPS: Gradient Descent Algorithm*

- Lets now go step by step to understand the **Gradient Descent algorithm:**
- **Step 1:** Initialize the weights (θ_0 and θ_1) with random values and calculate Error (SSE)
- **Step 2:** Calculate the gradient i.e. change in SSE when the weights (θ_0 and θ_1) are changed by a very small value from their original randomly initialized value. This helps us move the values of θ_0 and θ_1 in the direction in which SSE is minimized.
- **Step 3:** Adjust the weights with the gradients to reach the optimal values where SSE is minimized
- **Step 4:** Use the new weights for prediction and to calculate the new SSE
- **Step 5:** Repeat steps 2 and 3 till further adjustments to weights doesn’t significantly reduce the Error

* https://www.kdnuggets.com/2017/04/simple-understand-gradient-descent-algorithm.html

Gradient descent over multi-dimensional parameters

- Consider the object function f: $\Re^d \rightarrow \Re$ that takes any multi-dimensional vector $x{:}[x_1,\ldots,x_d]^T$ as its input. The gradient of $f(x)$ wrt x is defined by the vector of partial derivatives

$$\nabla_x f(x) = \left[\frac{\partial f(x)}{\partial x_1}, \cdots, \frac{\partial f(x)}{\partial x_d}\right]^T$$

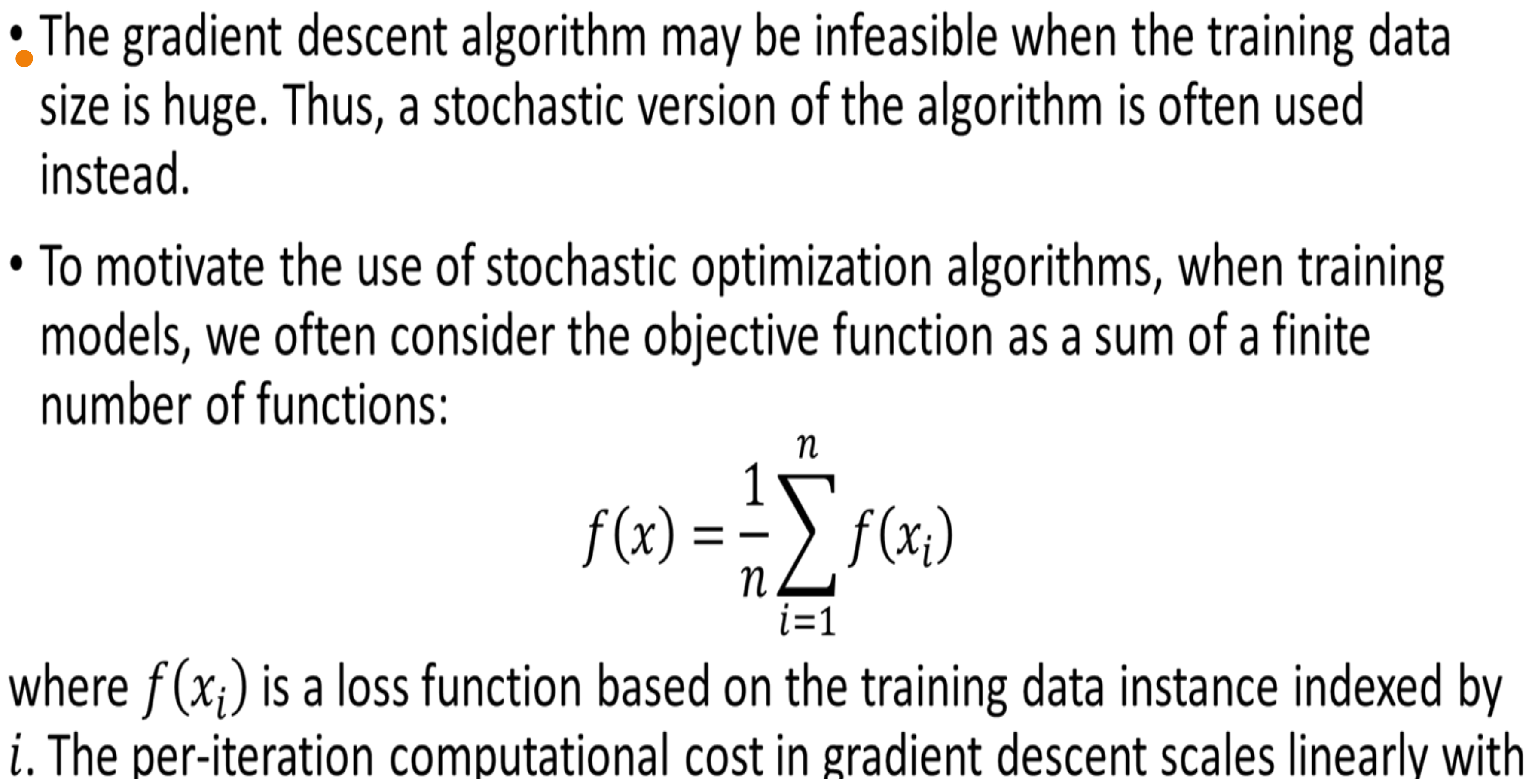
each element $\frac{\partial f(x)}{\partial x_i}$ of the gradient indicates the rate of change for f at the point x wrt the input x_i only.

- We can iteratively reduce the value of f with the following gradient decent update:

$$x := x - \boxed{?}\nabla f(x)$$

where η is positive and known as learning rate or step size.

STOCHASTIC GRADIENT DESCENT



- The gradient descent algorithm may be infeasible when the training data size is huge. Thus, a stochastic version of the algorithm is often used instead.
- To motivate the use of stochastic optimization algorithms, when training models, we often consider the objective function as a sum of a finite number of functions:

$$f(x) = \frac{1}{n}\sum_{i=1}^{n} f(x_i)$$

where $f(x_i)$ is a loss function based on the training data instance indexed by i. The per-iteration computational cost in gradient descent scales linearly with

STOCHASTIC GRADIENT DESCENT

- At each iteration, rather than computing the gradient $\nabla f(x)$, stochastic gradient descent randomly samples i at uniform and computes $\nabla f_i(x)$ instead. The insight is, stochastic gradient descent uses $\nabla f_i(x)$ as an unbiased estimator of $\nabla f(x)$.
- In a generalized case, at each iteration a mini-batch B that consists of indices for training data instances may be sampled at uniform with replacement. Similarly, we can use

$$\nabla f_{\mathrm{B}}(x) = \frac{1}{|\mathrm{B}|} \sum_{i \in \mathrm{B}} \nabla f_i(x)$$

to update x

$$x := x - ?\nabla f_{\mathrm{B}}(x)$$

where B is the cardinality of the mini batch.

LINEAR REGRESSION APPLICATION

- We have some data: as we observe the independent variables x_1 and x_2, we observe the dependent variable (or response variable) y along with it.
- In our dataset, we have 6 examples (or observations).

	x_1	x_2	y
1)	4	1	2
2)	2	8	-14
3)	1	0	1
4)	3	2	-1
5)	1	4	-7
6)	6	7	-8

(https://towardsdatascience.com/step-by-step-tutorial-on-linear-regression-with-stochastic-gradient-descent-1d35b088a843)

Model

- Basic linear deep network:

$$\hat{y} = w_1 x_1 + w_2 x_2 + b$$

- find the 'best' w and b values.

- The deep learning conventions w and b, which stand for weights and biases respectively.

Loss Function

Define loss function

- Let’s say at the end of this exercised, we’ve figured out our model to be

$$\hat{y} = 0.43x_1 - 0.21x_2 + 0.77$$

- How do we know if our model is doing well?
- We simply compare the predicted $\hat{y}$ and the observed y through a *loss function.* There are many ways to define the loss function but in this post, we define it as the squared difference between $\hat{y}$ and y.

$$L = (\hat{y} - y)^2$$

- Generally, the smaller the L, the better.

Weight Update equation

- Want to minimise the difference between $\hat{y}$ and y
- Use **stochastic gradient descent** optimization.
  - iteratively updating the values of w_1 and w_2 using the value of gradient and learning rate η, as in this equation:

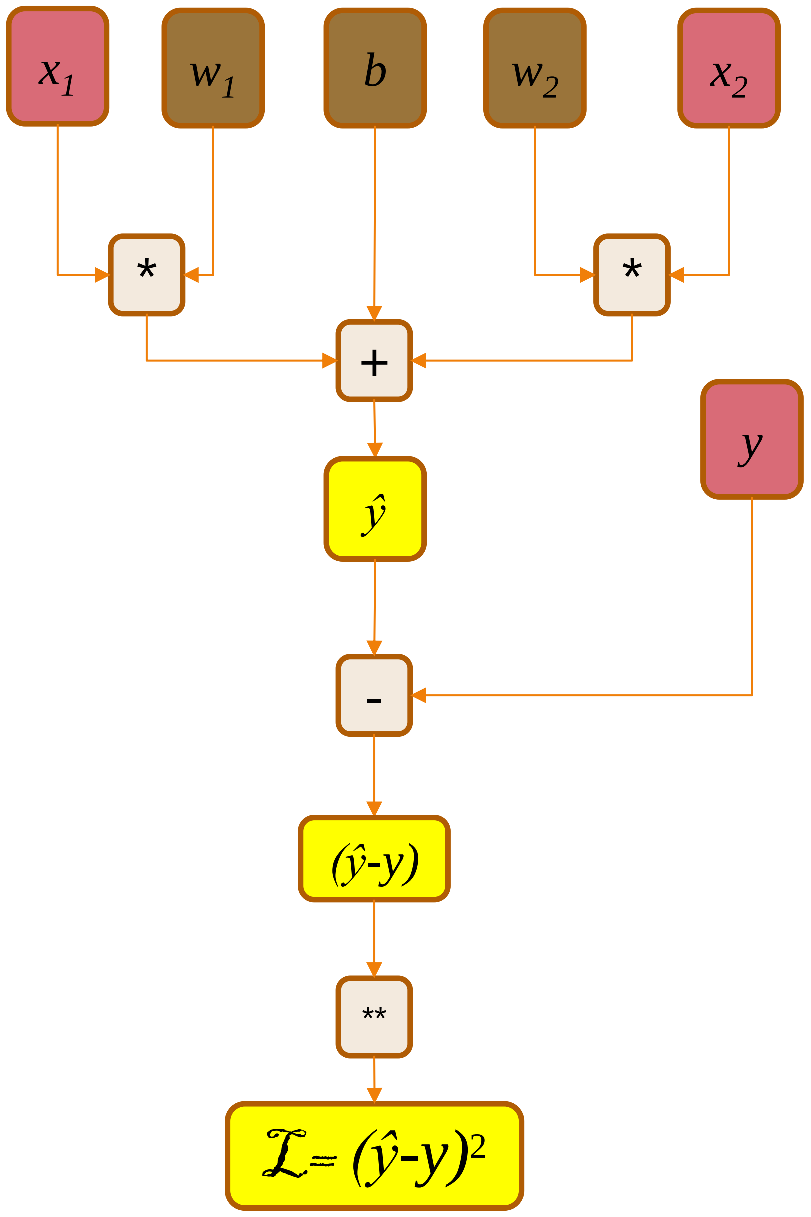
$$w_{\text{new}} = w_{\text{current}} - \eta \frac{\partial L}{\partial w_{\text{current}}}$$

- This algorithm tries to find the right weights by constantly updating them, bearing in mind that we are seeking values that minimize the loss function.

Algorithm

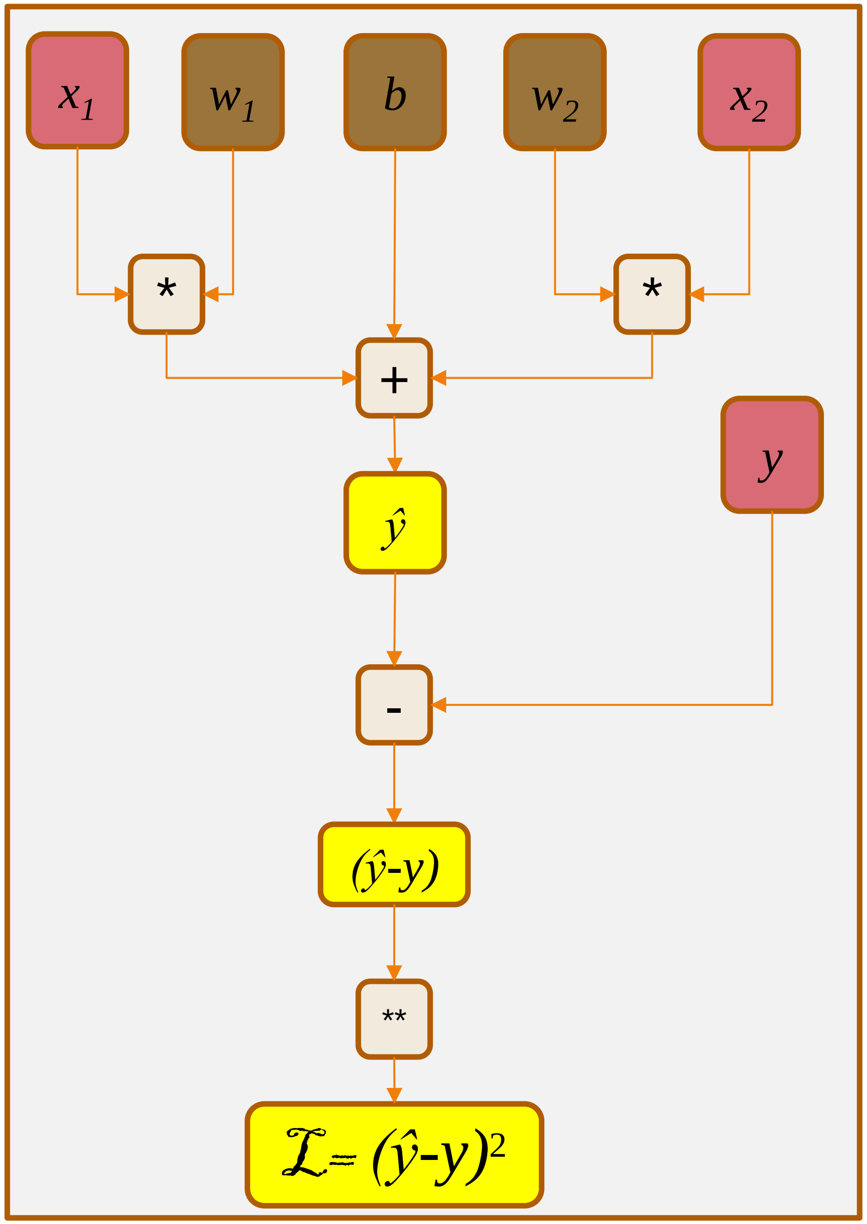
- The workflow for training our model is simple: forward propagation (or feed-forward or forward pass) and backpropagation.
- Training just means regularly updating the values of your weights

Computation Graph

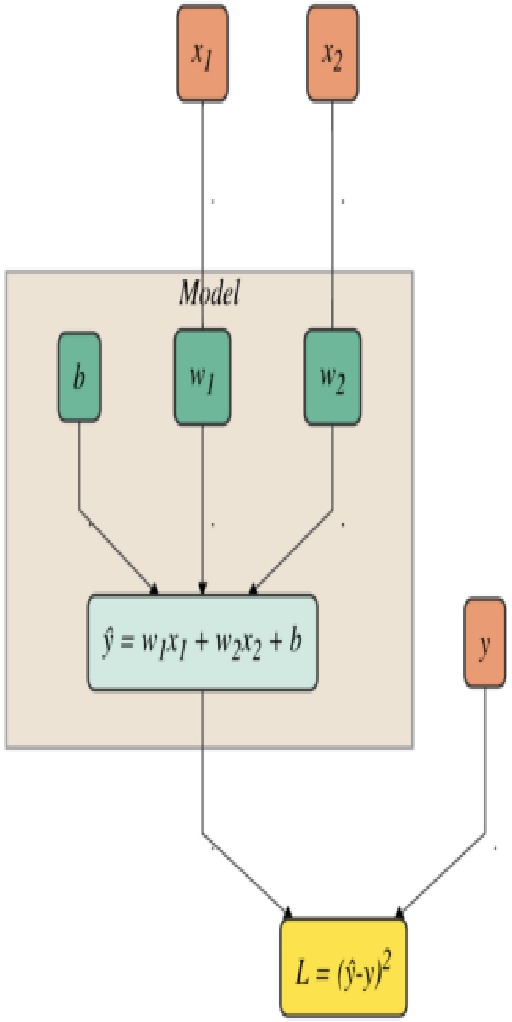

$$\begin{tikzcd}
x_1 \arrow[dr] & w_1 \arrow[d] & b \arrow[dd] & w_2 \arrow[d] & x_2 \arrow[dl] \\
& * \arrow[dr] & & * \arrow[dl] & \\
& & + \arrow[d] & & y \arrow[dd] \\
& & \hat{y} \arrow[d] & & \\
& & - \arrow[d] & & \arrow[ll] \\
& & (\hat{y}-y) \arrow[d] & & \\
& & ** \arrow[d] & & \\
& & \mathcal{L} = (\hat{y}-y)^2 & &
\end{tikzcd}$$

- Simple linear regression
  - orange—the placeholders (x_1, x_2 and y),
  - dark green—the weights and bias (w_1, w_2 and b),
  - light green—the operators *, +, -, **
  - yellow—the model: (ŷ) , (ŷ-y), loss function (L).

Computation Graph (Simplified)

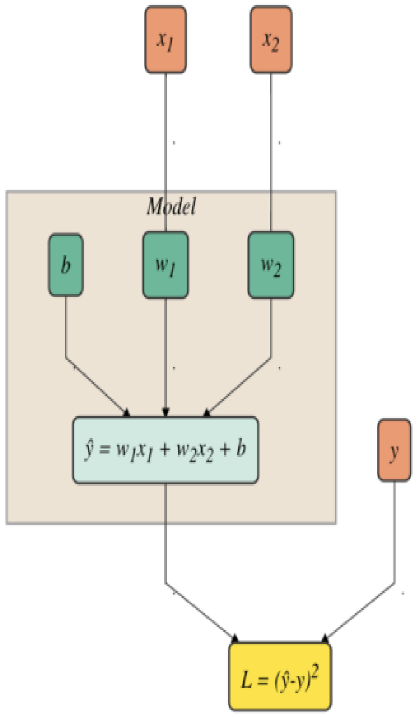


Simplified Computation Graph



STEP 0. Build computation graph

- To keep track of all the values, we build a ‘computation graph’ that comprises nodes color-coded in
  - orange—the placeholders (x_1, x_2 and y),
  - dark green—the weights and bias (w_1, w_2 and b),
  - light green—the model (ŷ) connecting w_1, w_2, b, x_1 and x_2, and
  - yellow—the loss function (L) connecting the ŷ and y.



For forward propagation, you should read this graph from top to bottom and for backpropagation bottom to top.

STEP 1. Forward Propagation

Initialize weights (one-time)

- Since gradient descent is all about updating the weights, we need them to start with some values, known as initializing weights.
- Here we initialized the weights and bias as follows:

$$w_1 = -0.017$$

$$w_2 = -0.048$$

$$b = 0$$

In this example, we initialized the weights by using truncated normal distribution and the bias with 0.

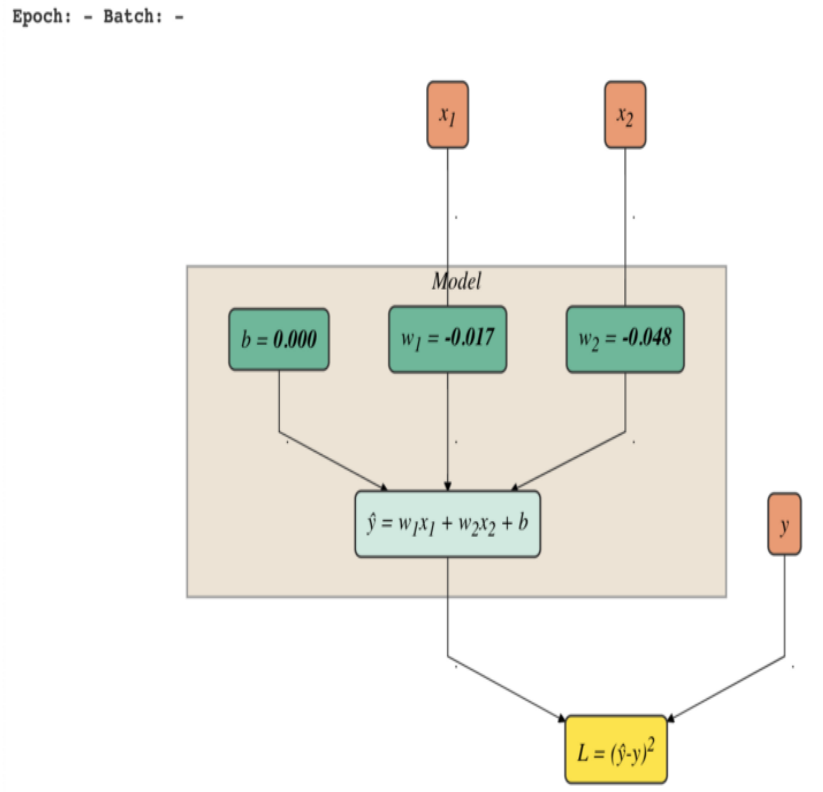


Fig. 1: Weights initialized (dark green nodes)

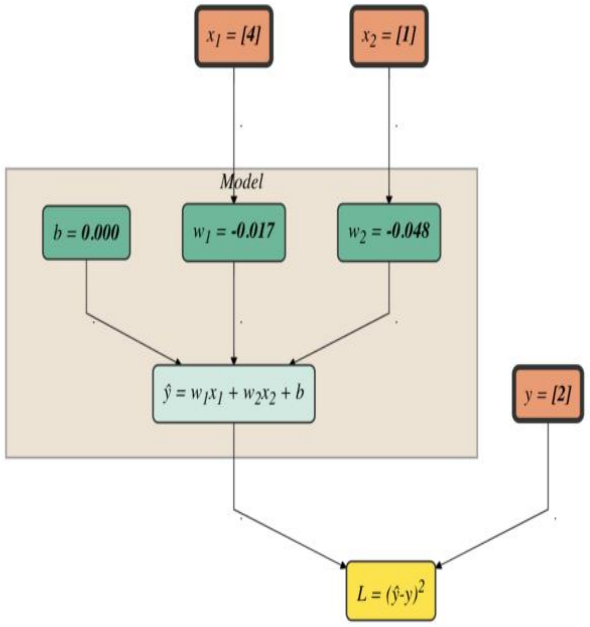
STEP 2. Forward Propagation: Feed data

- We set the batch size to be 1 and we feed in this first batch of data.
- **Batch and batch size:** We can divide our dataset into smaller groups of equal size. Each group is called a **batch** and consists of a specified number of examples, called **batch size**. If we multiply these two numbers, we should get back the number of observations in our data.
- Here, our dataset consists of 6 examples and since we defined the batch size to be 1 in this training, we have 6 batches altogether.

```
Epoch: 1/1 Batch: 1/6
Forward propagation
```

$$x_1 = 4$$
$$x_2 = 1$$
$$y = 2$$

Eqn. 1: First batch of data fed into model

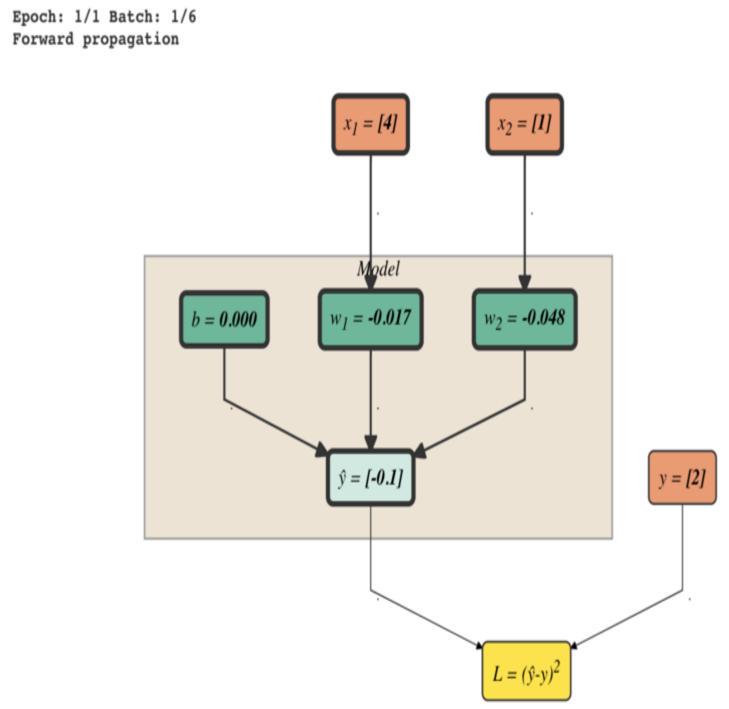


STEP 3. Forward Propagation

Compute ŷ

- Now that we have the values of x_1, x_2, w_1, w_2 and b ready, let's compute ŷ.

$$\begin{aligned}\hat{y} &= w_1x_1 + w_2x_2 + b \\ &= (-0.017)\cdot 4 + (-0.048)\cdot 1 + 0.000 \\ &= -0.116\end{aligned}$$



- The value of $\hat{y}$ (=–0.1) is reflected in the **light green node** below:

L computed (yellow node)

$$
\begin{aligned}
L &= (\hat{y} - y)^2 \\
&= (-0.116 - 2)^2 \\
&= 4.48
\end{aligned}
$$

Eqn. 2: Compute the loss

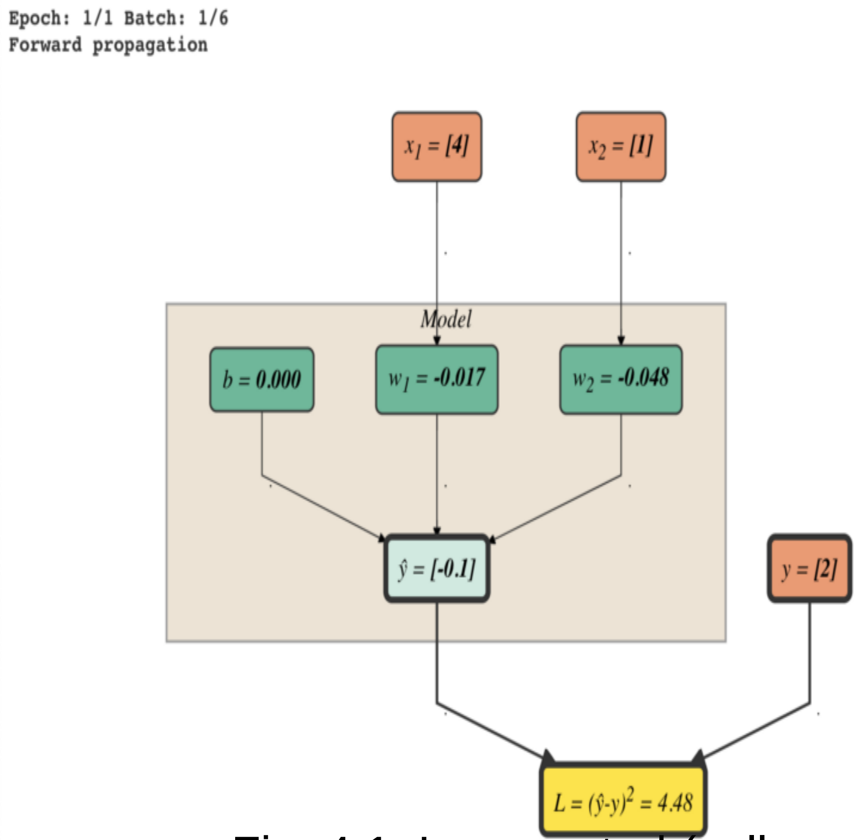
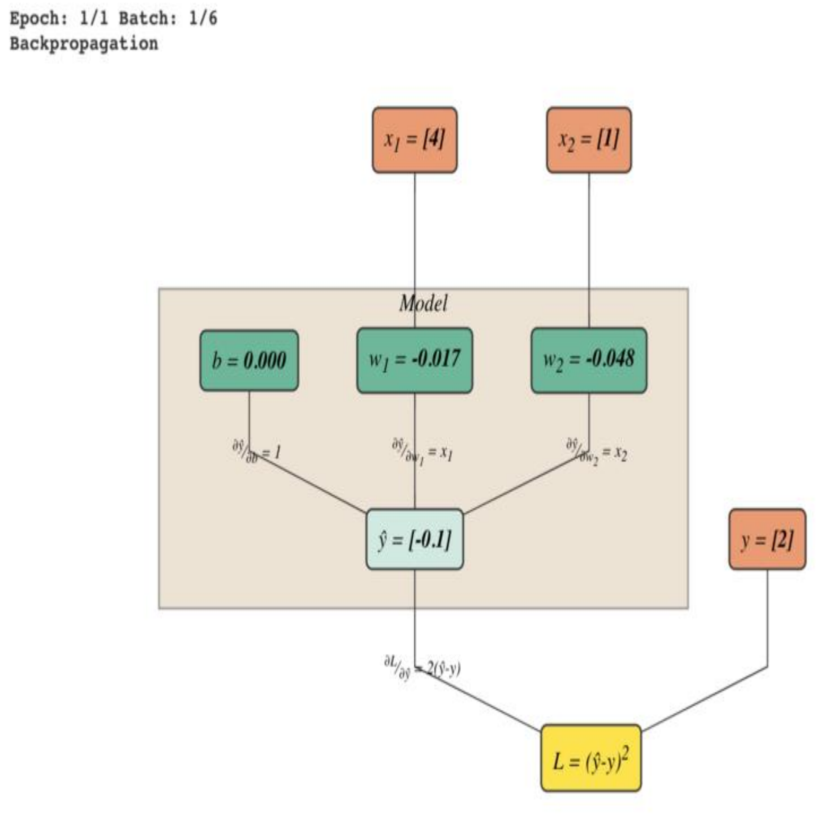


Fig. 4.1: L computed (yellow node)

It is a common practice to log the loss during training, together with other information like the epoch, batch and time taken.

STEP 5: Backpropagation: Compute partial differentials

- Before we start adjusting the values of the weights and bias w_1, w_2 and b, let's first compute all the partial differentials. These are needed later when we do the



```
Epoch: 1/1 Batch: 1/6
Backpropagation
```

$x_1 = [4]$
$x_2 = [1]$
Model
$b = 0.000$
$w_1 = -0.017$
$w_2 = -0.048$
$\partial\hat{y}/\partial b = 1$
$\partial\hat{y}/\partial w_1 = x_1$
$\partial\hat{y}/\partial w_2 = x_2$
$\hat{y} = [-0.1]$
$y = [2]$
$\partial L/\partial\hat{y} = 2(\hat{y}-y)$
$L = (\hat{y}-y)^2$

Fig. 5: Indicated partial differentials to the relevant edges on the graph

- Namely, we compute **all possible paths** leading to every *w* and *b* only, because these are the only variables which we are interested in updating.
- From Fig. 5, we see that there are 4 edges that we labeled with the partial differentials.

$\hat{y} = w_1 x_1 + w_2 x_2 + b \Rightarrow$ Model

$L = (\hat{y} - y)^2 \Rightarrow$ Loss

- Recall the equations for the model and loss function:

- The partial differentials are as follows:

L (**yellow**)—*ŷ* (**light green**): $\frac{\partial L}{\partial \hat{y}} = 2(\hat{y} - y)$

ŷ (**light green**)—*b* (**dark green**): $\frac{\partial \hat{y}}{\partial b} = 1$

ŷ (**light green**)—w_1 (**dark green**): $\frac{\partial \hat{y}}{\partial w_1} = x_1$

ŷ (**light green**)—w_2 (**dark green**): $\frac{\partial \hat{y}}{\partial w_2} = x_2$

STEP 6. Backpropagation: Update weights

- Observe the dark green nodes in Fig. 6 below. We see three things:
  - i) b changes from 0.000 to 0.212
  - ii) w_1 changes from –0.017 to 0.829
  - iii) w_2 changes from –0.048 to 0.164

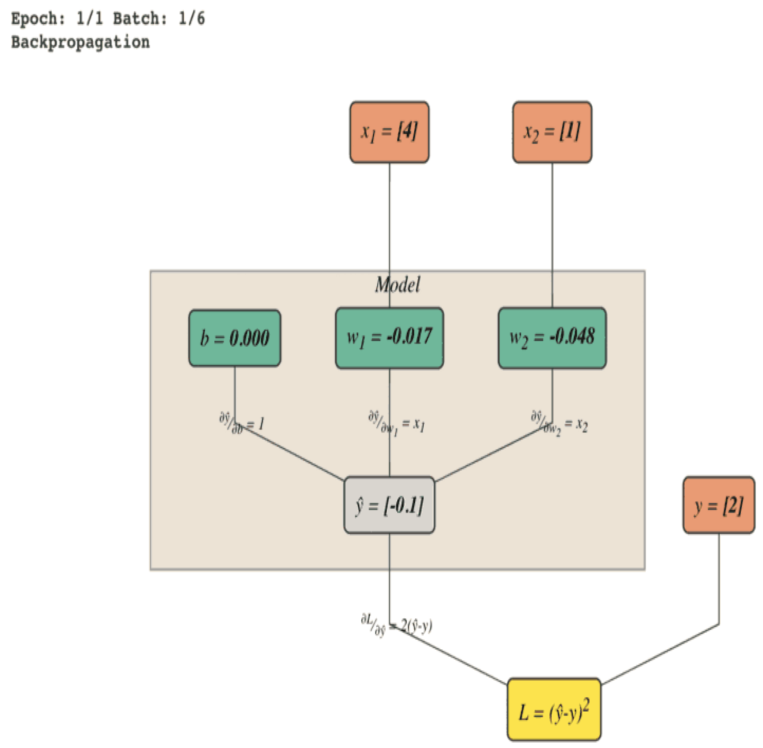


Fig. 6: Updating the weights and bias (dark green nodes)

- This is stochastic gradient descent—updating the weights using backpropagation, making use of the respective gradient values.
- Let’s first focus on updating b. The formula for updating b is

$$b' = b - \eta \frac{\partial L}{\partial b} \Rightarrow \text{Stochastic gradient descent update for b}$$

where

b—current value

b’—value after update

η —learning rate, set to 0.05

∂L/∂b—gradient i.e. partial differential of *L* w.r.t. *b*

- To get the gradient, we need to multiply the paths from L leading to b using chain rule:

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial b}$$

- We would require the current batch values of *x*, *y*, *ŷ* and the partial differentials so let’s just place them below for easy reference:

$$\frac{\partial L}{\partial \hat{y}} = 2(\hat{y} - y)$$

$$\frac{\partial \hat{y}}{\partial w_1} = x_1$$

$$\frac{\partial \hat{y}}{\partial w_2} = x_2$$

$$\frac{\partial \hat{y}}{\partial b} = 1$$

$$\Longrightarrow$$

Values from current batch and the predicted ŷ

$$x_1 = 4$$

$$x_2 = 1$$

$$y = 2$$

$$\hat{y} = -0.116$$

Update weights

- Using the stochastic gradient descent equation and plucking in all the values gives us

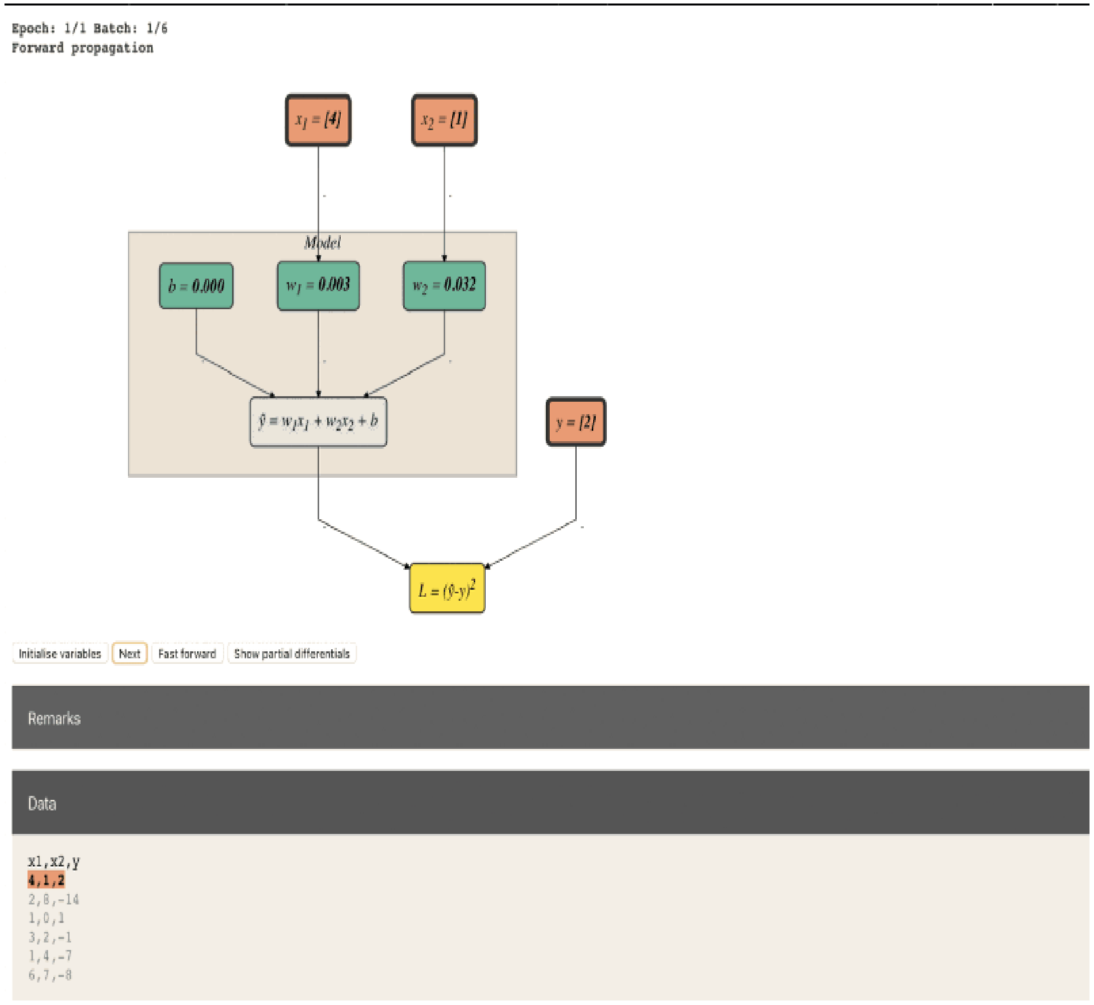
$$\begin{aligned} b' &= b - \eta \frac{\partial L}{\partial b} \\ &= b - \eta \left(\frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial b}\right) \\ &= b - \eta [2(\hat{y} - y) \cdot 1] \\ &= 0.000 - 0.05[2(-0.116 - 2) \cdot 1] \\ &= 0.212 \end{aligned}$$

That's it for updating *b*! We are left with updating w_1 and w_2, which we update in a similar fashion.

$$\begin{aligned} w_1' &= w_1 - \eta \frac{\partial L}{\partial w_1} \\ &= w_1 - \eta \left(\frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w_1}\right) \\ &= w_1 - \eta [2(\hat{y} - y) \cdot x_1] \\ &= -0.017 - 0.05[2(-0.116 - 2) \cdot 4] \\ &= 0.829 \end{aligned}$$

$$\begin{aligned} w_2' &= w_2 - \eta \frac{\partial L}{\partial w_2} \\ &= w_2 - \eta \left(\frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w_2}\right) \\ &= w_2 - \eta [2(\hat{y} - y) \cdot x_2] \\ &= -0.048 - 0.05[2(-0.116 - 2) \cdot 1] \\ &= 0.164 \end{aligned}$$

- Now we need to iterate the above-mentioned steps to the other 5 batches, namely examples 2 to 6.



```
Epoch: 1/1 Batch: 1/6
Forward propagation
```

```
x1,x2,y
4,1,2
2,8,-14
1,0,1
3,2,-1
1,4,-7
6,7,-8
```

End of epoch

- We complete 1 epoch when the model has iterated through all the batches once. In practice, we extend the epoch to more than 1.
- One epoch is when our setup has seen all the observations in our dataset once. But one epoch is almost always never enough for the loss to converge. In practice, this number is manually tuned.
- At the end of it all, you should get a final model, ready for inference, say:

$$\hat{y} = 0.43x_1 - 0.21x_2 + 0.77$$

Improve training

- One epoch is never enough for a stochastic gradient descent optimization problems. Remember that our first loss value is at 4.48. If we increase the number of epochs, which means just increasing the number of times we update the weights and biases, we can converge it to a satisfactory low.
- Below are the things you can improve the training:
  - Extend training to more than 1 epoch
  - Increase batch size
  - Change optimizer
  - Adjust learning rate (changing the learning rate value or using learning rate schedulers)
  - Hold out a train-validation-test set