Deep Network Developments

Lecture #6

Viktor Varga
Department of Artificial Intelligence, ELTE IK

Requirements

⊗

The content of the slides marked by this symbol **will not be included in the exams / tests.**

Last week - Supervised learning

Given: The training sample, a set of (input, label) pairs

$$\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$$

$$x \in \boldsymbol{X} \subset \mathbb{R}^n,\ y \in \boldsymbol{Y} \subset \mathbb{R}^k$$

Task: The estimation of the label (the expected output) from the input

I.e., we search for a (hypothesis-)function h_θ, for which:

$$h_\theta(x) = \hat{y} \approx y$$

Last week - Two main tasks in supervised learning

Regression: Continuous labels (The label set is infinite)

$|Y| = \infty$ **Example:** Number of cars or the age of a person

Classification: Discrete labels (The label set is finite)

$|Y| < \infty$ **Example:** Categorization of examples

- What is the profession of the person in the image?

Last week - Least squares solution in one step

The normal equation - an **exact solution** to the least squares problem

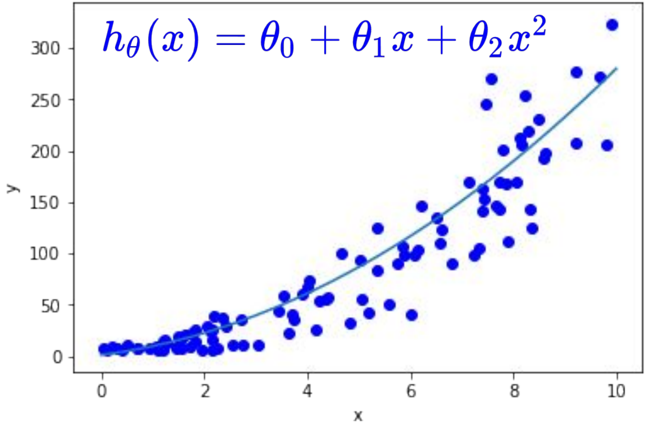
$$\theta = (X^T X)^{-1} X^T y$$

The regularized (L2) least squares solution:

$$\theta = (X^T X + \boxed{\lambda I})^{-1} X^T y$$

Last week - Polynomial regression

Hypothesis function



Univariate:

$$h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \ldots$$

Multivariate, e.g.,:

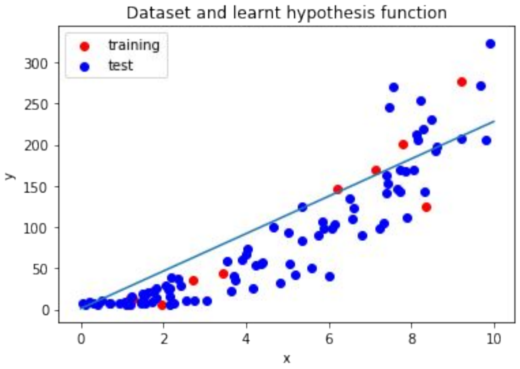
$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1^2 + \theta_4 x_2^2 + \theta_5 x_1 x_2 + \ldots$$

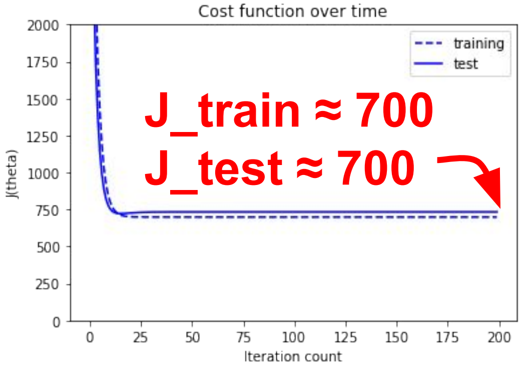
Loss function (MSE):

$$J(\theta) = \frac{1}{2m} \sum_{j=1}^{m} (\overbrace{h_\theta(x^{(j)})}^{\hat{y}^{(j)}} - y^{(j)})^2$$

Last week - Underfitting / “just right” / overfitting

$$h_\theta(x) = \theta_0 + \theta_1 x$$

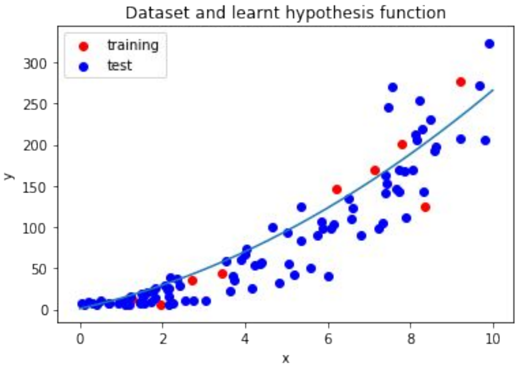


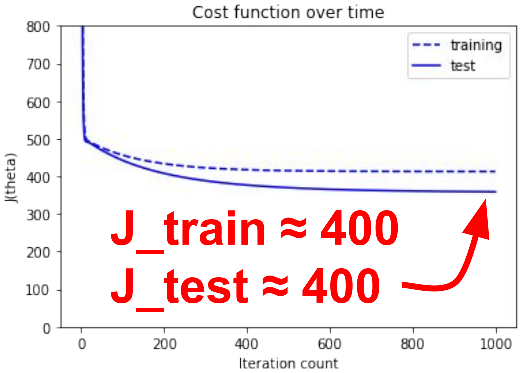


J_train ≈ 700

J_test ≈ 700

$$h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2$$

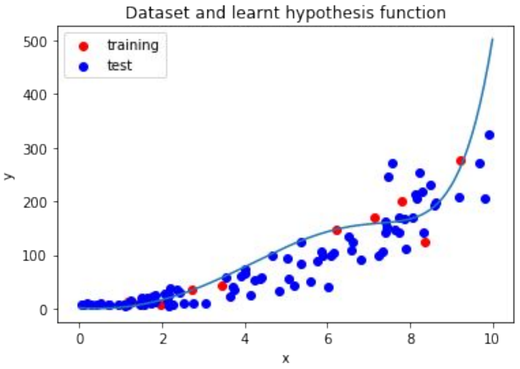


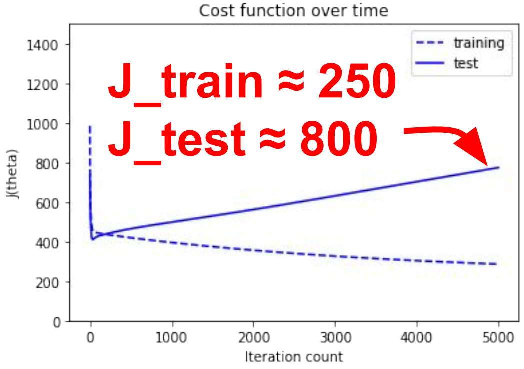


J_train ≈ 400

J_test ≈ 400

$$h_\theta(x) = \theta_0 + \theta_1 x + \cdots + \theta_9 x^9$$



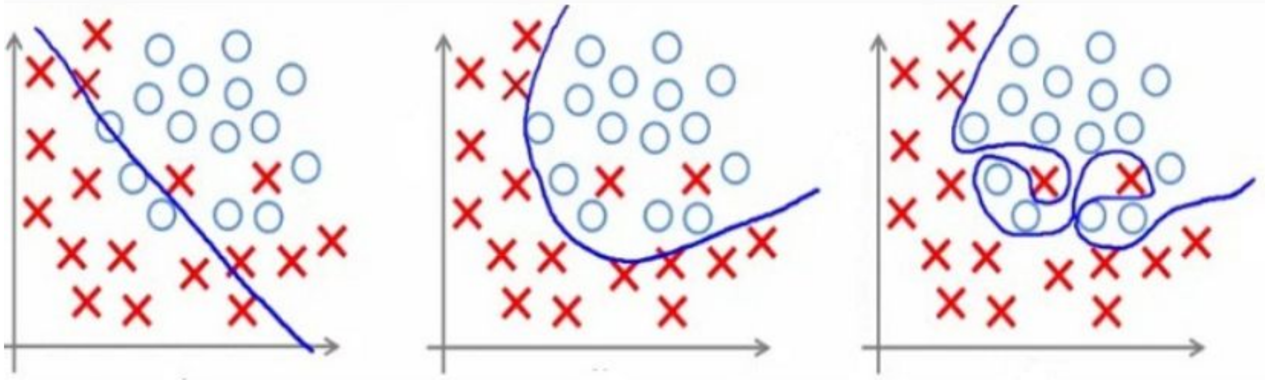


J_train ≈ 250

J_test ≈ 800

Image source: Andrew Ng Machine learning video series

Last week - Under- and overfitting in classification



Under-fitting

(too simple to explain the variance)

Appropriate-fitting

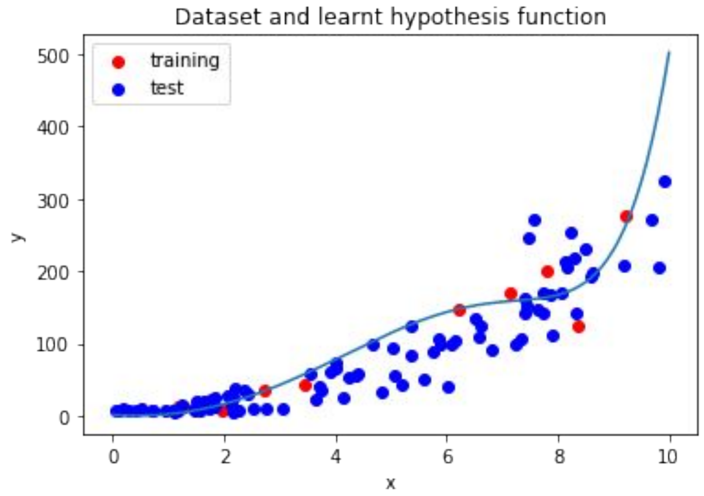
Over-fitting

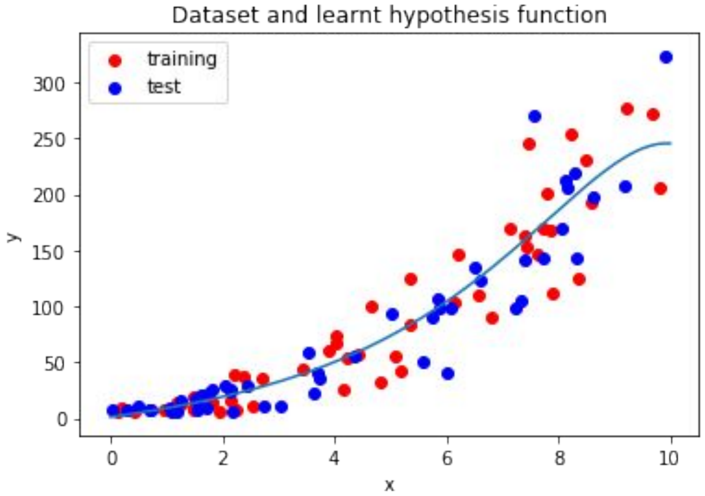
(forcefitting -- too good to be true)

Last week - How to deal with **overfitting**?

Obtain more training data!

$$h_\theta(x) = \theta_0 + \theta_1 x + \cdots + \theta_9 x^9$$





$|X_{train}| = 10$

J_train ≈ 250
J_test ≈ 800

$|X_{train}| = 50$

J_train ≈ 400
J_test ≈ 400

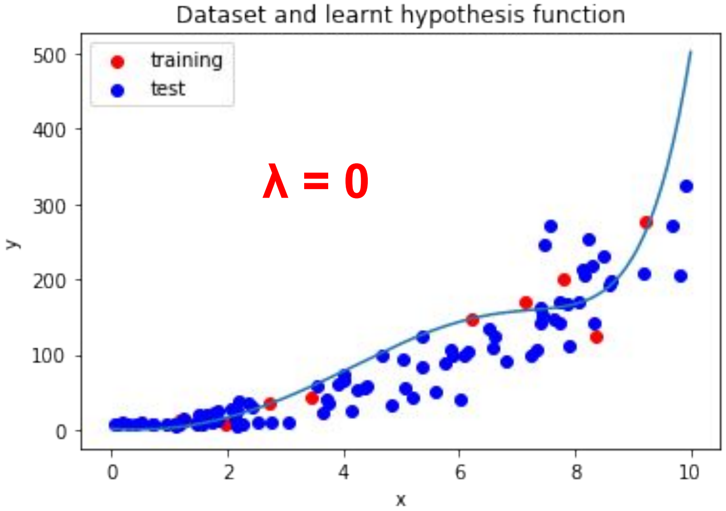
Last week - How to deal with **overfitting**?

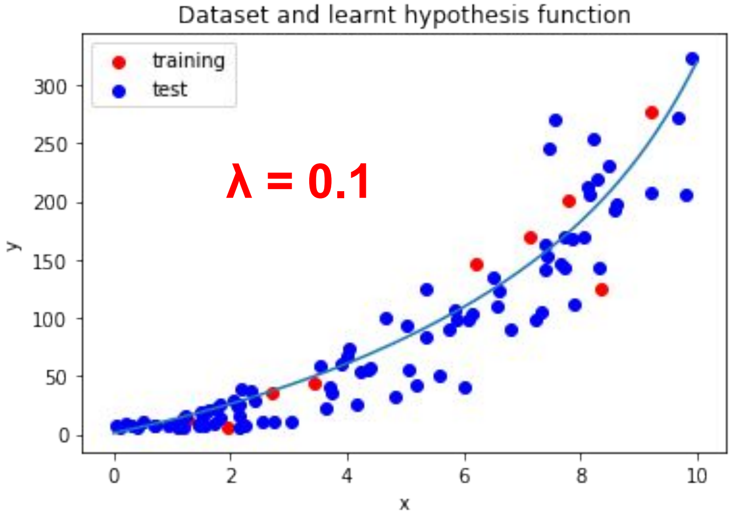
L2 regularization

$$|X_{train}| = 10$$

$$h_\theta(x) = \theta_0 + \theta_1 x + \cdots + \theta_9 x^9$$

$$J(\theta) = \frac{1}{2m}\sum_{j=1}^{m}(h(x)^{(j)} - y^{(j)})^2 + \boxed{\lambda \sum_{i=1}^{n} \theta_i^2}$$



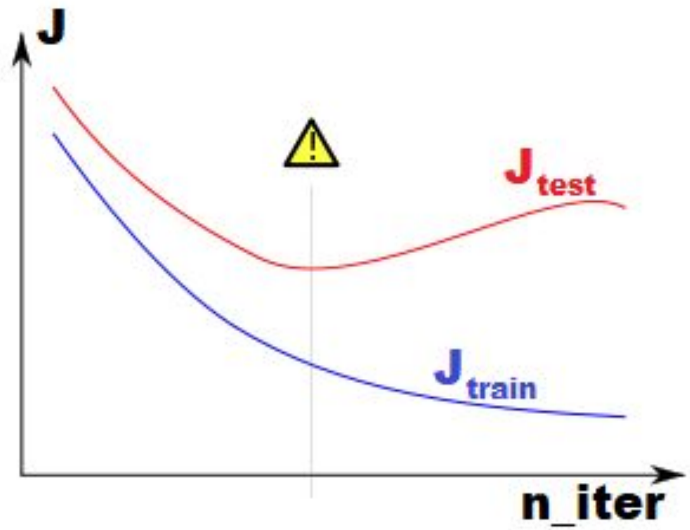


Last week - How to deal with **overfitting**?

Early stopping (The incorrect way)

New loop condition: Loop, while J_{test} keeps reducing.

```
~~repeat until convergence~~ {
    for i ← 1 . . . n   {
        $grad_i = \frac{\partial}{\partial \theta_i} J(\theta)$
    }
    for i ← 1 . . . n   {
        $\theta_i = \theta_i - \alpha \; grad_i$
    }
}
```



Last week - Splitting the sample

Splitting the sample into three sets:

New set

Training set, **validation set,** test set

We will use the validation set to optimize the following parameters:

- *Learning rate (alpha)*
- *Polynomial degree, or neural network architecture (layers, number of neurons, etc.)*
- *Number of iterations for the gradient method (early stopping)*
- *...*

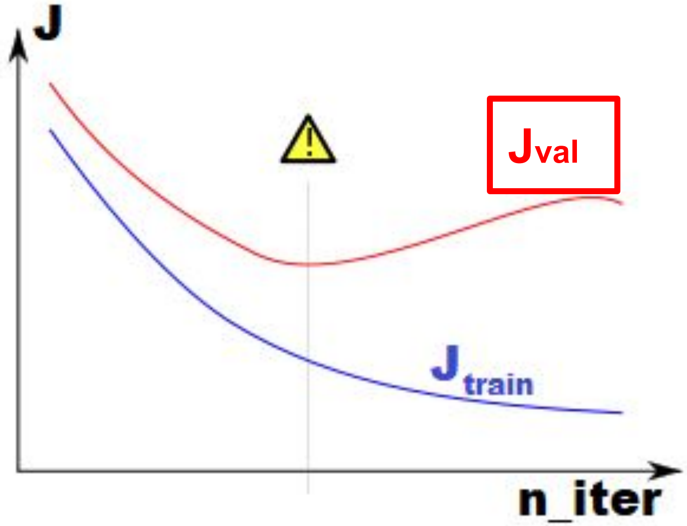
→ **Hyperparameters…**

Last week - How to deal with **overfitting**?

Early stopping (The correct way)

New loop condition: Loop, while J_{val} keeps reducing.

```
~~repeat until convergence~~ {
    for i ← 1 . . . n  {
        $grad_i = \frac{\partial}{\partial \theta_i} J(\theta)$
    }
    for i ← 1 . . . n  {
        $\theta_i = \theta_i - \alpha \; grad_i$
    }
}
```



Last week - Model training procedure

The task: $\psi^*, \theta^* = argmin_\psi \; argmin_\theta \; J_\psi(\theta)$

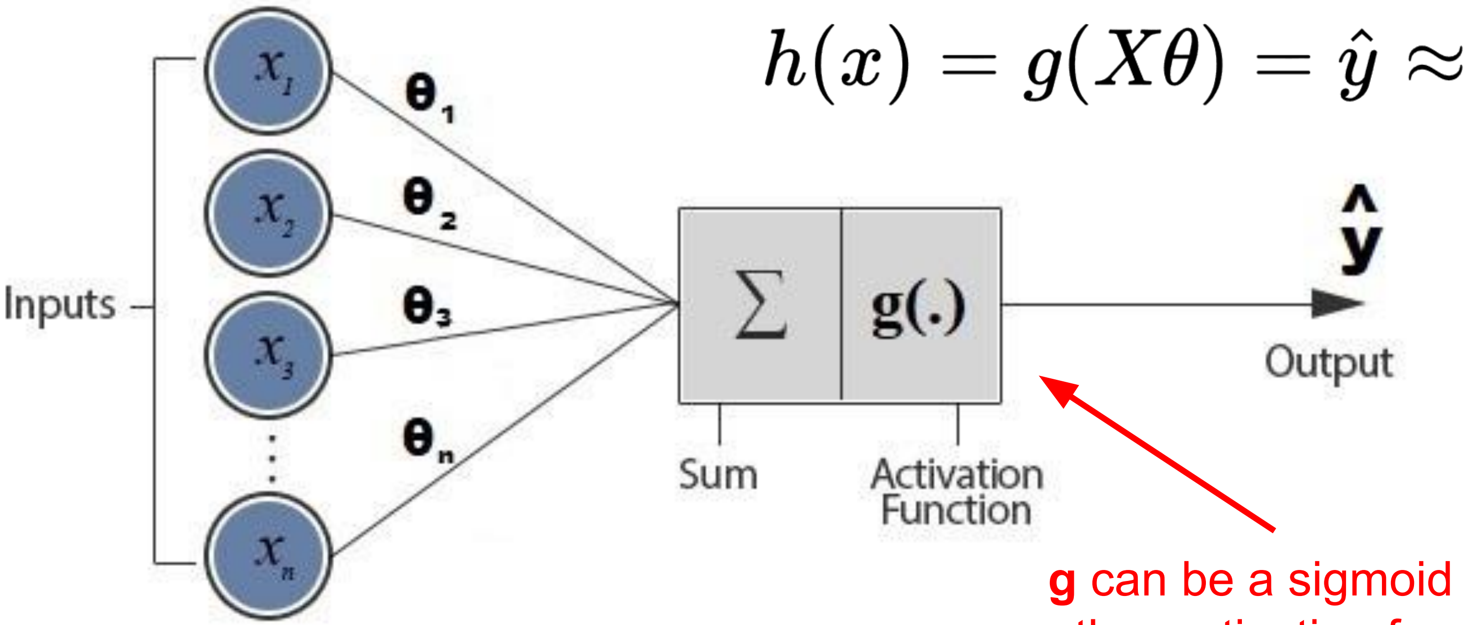
1) Select a new hyperparameter configuration **Ψ**.
2) Optimize the model parameters (**θ**) on the **training set** with gradient descent.
3) Evaluate the trained model on the **validation set**, then GOTO 1

Finally:

- **Ψ*** := The hyperparameter configuration with the best performance on the validation set.
- **θ*** := The trained model parameters with hyperparameters **Ψ***.
- Evaluate model with parameters **θ*** and hyperparameters **Ψ*** on the **test set.**

Last week - The artificial neuron model

$$h(x) = g(X\theta) = \hat{y} \approx y$$

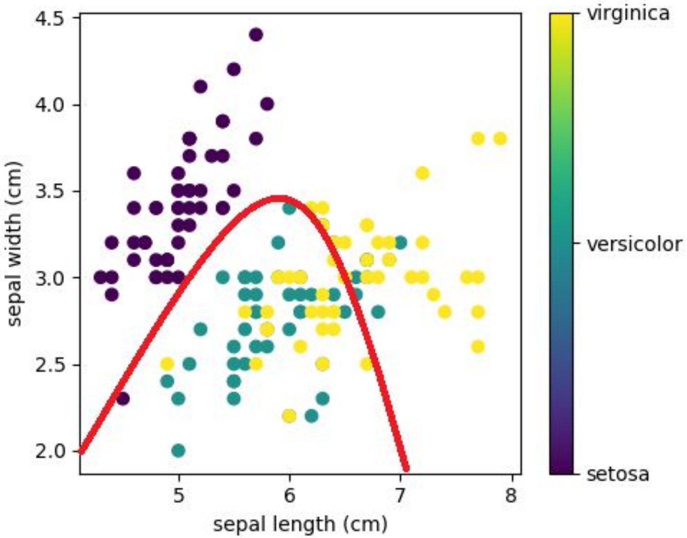


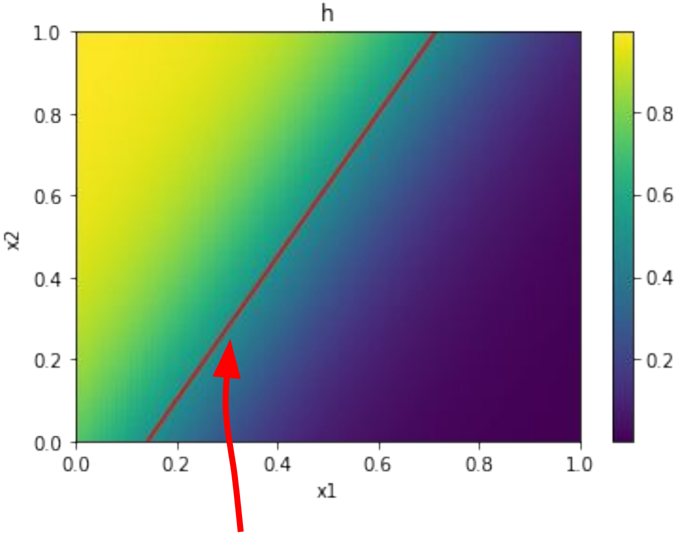
g can be a sigmoid or other activation functions

An artificial neuron is a (multivariate) **logistic regression if g is sigmoid!**

Last week - Multilayer Perceptron (MLP)

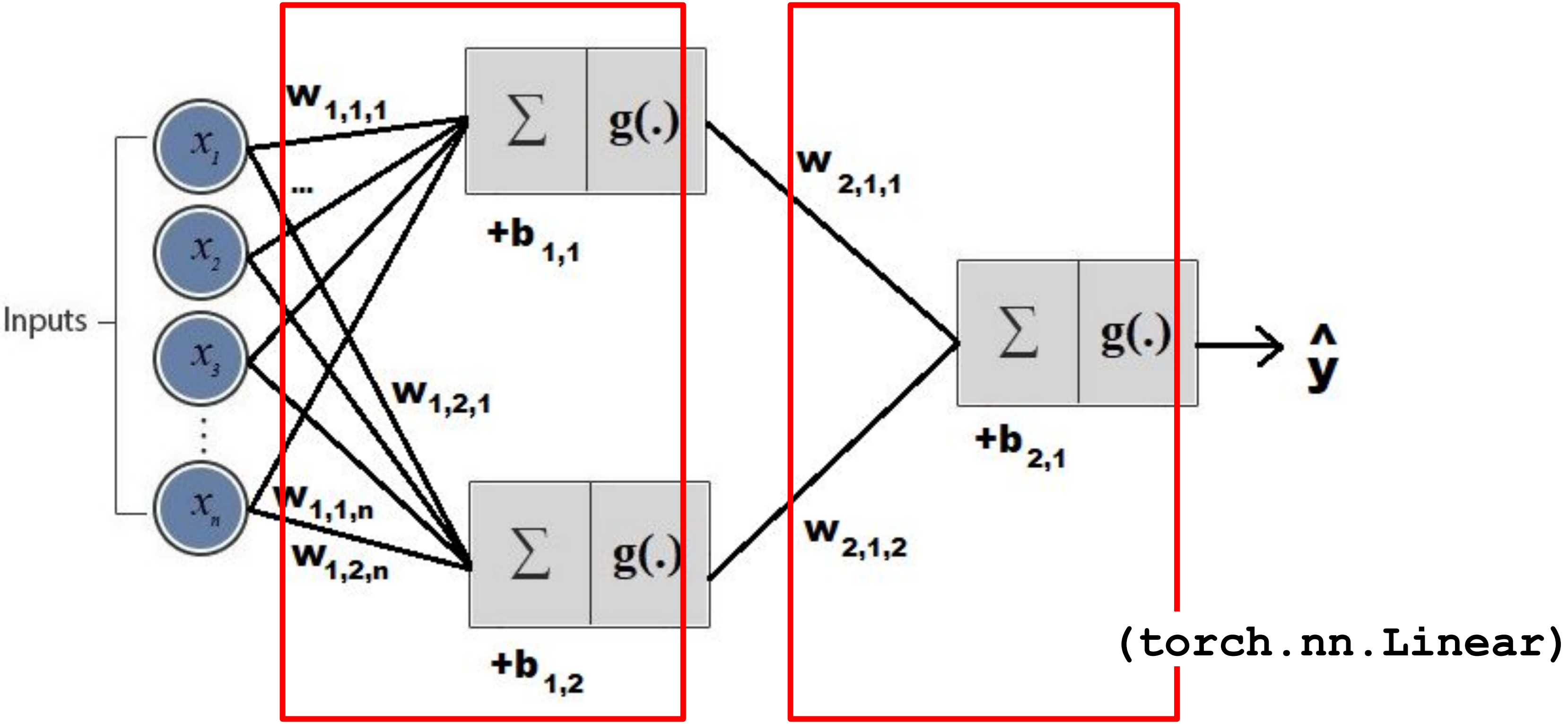
Most tasks cannot be solved by a single artificial neuron.





A linear decision boundary represented by **a single neuron is not capable of solving linearly inseparable problems**.

Last week - Multilayer Perceptron (MLP)

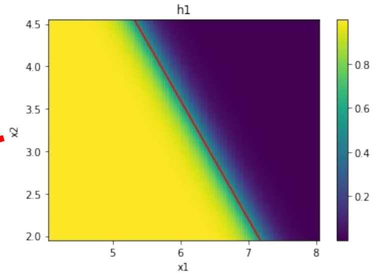


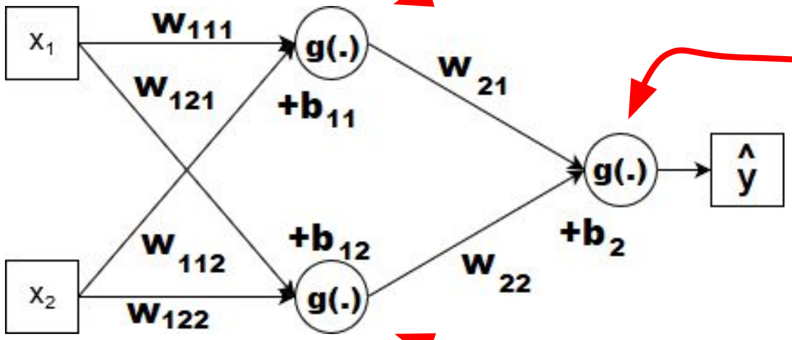
(torch.nn.Linear)

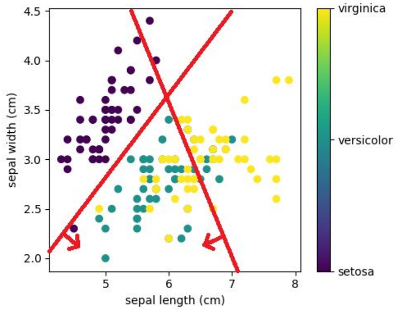
Fully connected / dense (neuron) **layers**

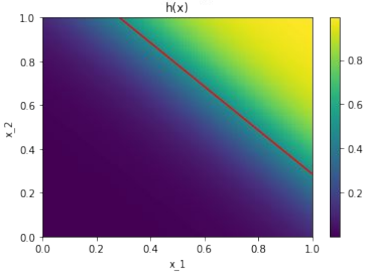
Last week - The expressive power of neural networks

$$w_{111} = -7,\ w_{121} = -5,\ b_{11} = 60$$





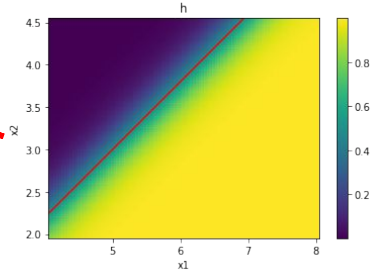


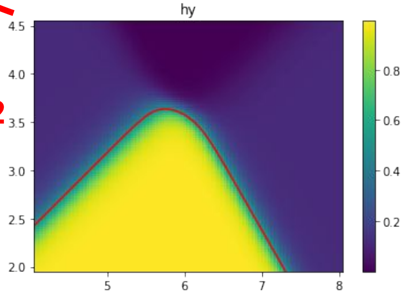


h2

h1

$$w_{21} = 7,\ w_{22} = 7,\ b_2 = -9$$





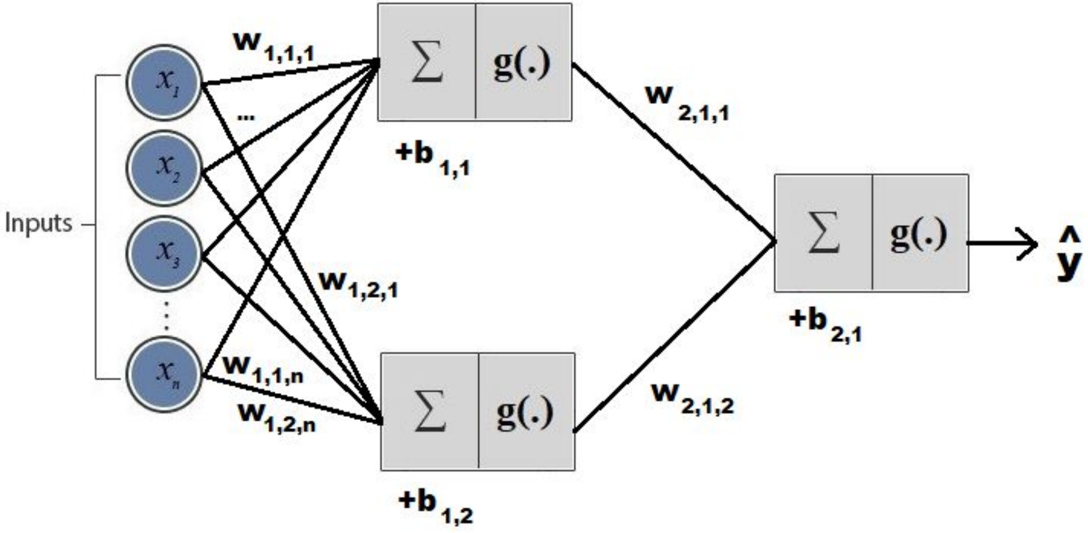
x2

x1

The weights were set manually here…

$$w_{112} = 4,\ w_{122} = -5,\ b_{12} = -5$$

Last week - Multilayer Perceptron (MLP)



$$W_1 \in \mathbb{R}^{2\times n}$$
$$b_1 \in \mathbb{R}^2$$

$$W_2 \in \mathbb{R}^{1\times 2}$$
$$b_2 \in \mathbb{R}^1$$

New notation: Theta is the set of all weight matrices and bias vectors.
(i.e., the parameters)

$$\Theta = \{W_1, b_1, W_2, b_2\}$$

Last week - Multilayer Perceptron (MLP)

The hypothesis function of a two-layer MLP neural network:

$$h(x) = g_2(W_2\, \underbrace{g_1(W_1 x + b_1)}_{\textbf{The output of the first layer}} + b_2) = \hat{y} \approx y$$

- **We always need intermediate activation functions** (nonlinearities) **in MLPs**
- **The last activation function depends on the task**

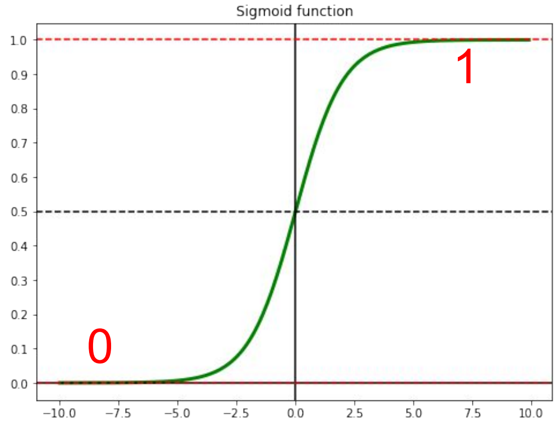
Loss functions:

- **Regression:** MSE
- **Classification** (binary)**:** Logistic loss / Binary Cross-entropy (BCE)

Last week - Activation functions

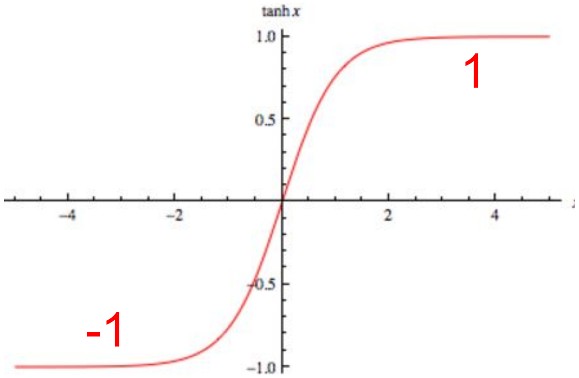
Popular activation functions

sigmoid



$$g(z) = \frac{1}{1+e^{-z}}$$

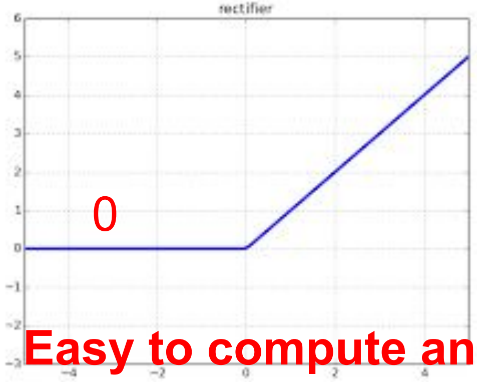
tanh



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

ReLU

(Rectified Linear Unit)



Easy to compute and almost always works well

$$g(z) = \mathrm{ReLU}(z) = \max(0, z)$$

Last week - Training an MLP

“Batch” gradient descent

We average the loss over data points in the entire training set.

For example:

$$J(\theta) = \frac{1}{2m}\sum_{j=1}^{m}(h_\theta(x^{(j)}) - y^{(j)})^2$$

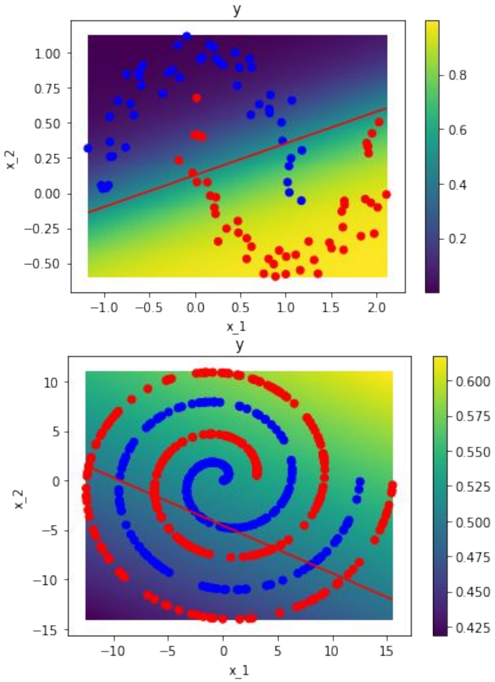
```
repeat until convergence   {
   for $\forall\ \theta \in \Theta$   {
      $grad_\theta = \frac{\partial}{\partial\theta} J(\Theta)$
   }
   for $\forall\ \theta \in \Theta$   {
      $\theta = \theta - \alpha\ grad_\theta$
   }
```

Usually more efficient: **Stochastic Gradient Descent** (SGD) **algorithm**
Compute gradients on a randomly selected sub-sample of the dataset (a mini-batch)
→ **Approximate, but much faster computation of the gradients.**

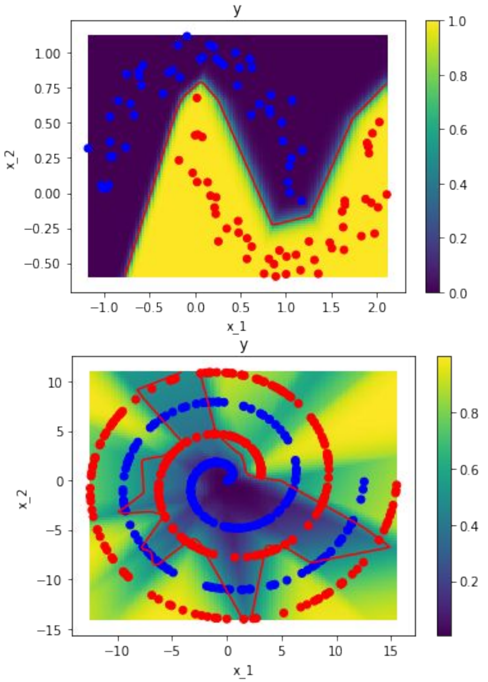
These weights were found using gradient descent.

Last week - The expressive power of neural networks

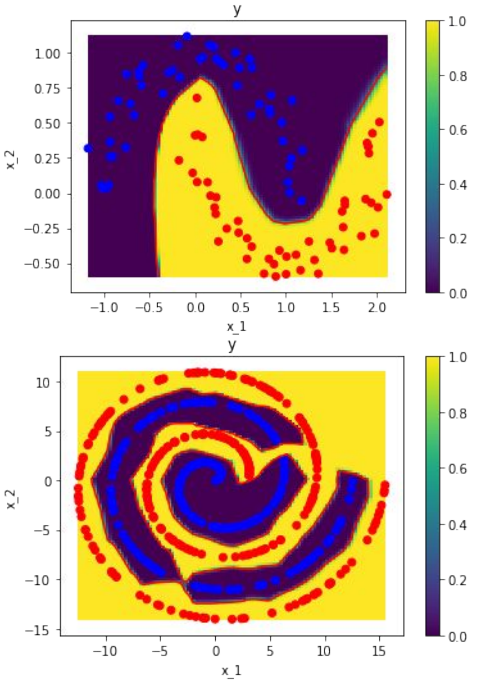
Logistic regression



MLP, 2 layers, 20+1 neurons



MLP, 4 layers, 20+20+20+1 neurons



Last week - Example tasks with multiple labels

A regression task with multiple label variables:

x_1: Weight of a patient

x_2: Age of a patient

x_3: Sex of a patient

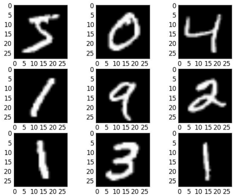
→

y_1: Cholesterol levels of the patient

y_2: Blood sugar levels of the patient

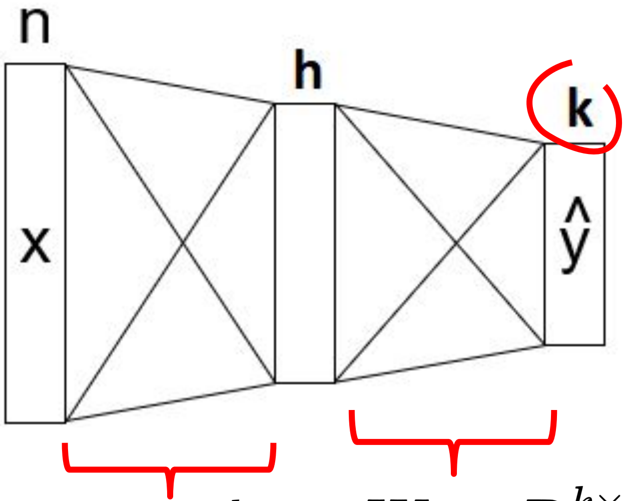
A classification task with more than two categories (multi-class):

x_1, ..., **x_{784}** : Brightness of pixels → **y_1**, ..., **y_{10}**: Probabilities of the digits from 0 to 9 being in the image



Last week - Multiple label variables

$$h(x) = g_2(W_2\, g_1(W_1 x + b_1) + b_2) = \hat{y} \approx y$$



$W_1 \in \mathbb{R}^{h \times n}$

$b_1 \in \mathbb{R}^{h}$

$W_2 \in \mathbb{R}^{k \times h}$

$b_2 \in \mathbb{R}^{k}$

Let y be a vector, similarly to x!

$$\Theta = \{W_1, b_1, W_2, b_2\}$$

Last week - Multiple label variables, **regression**

$$h(x) = g_2(W_2\, g_1(W_1 x + b_1) + b_2) = \hat{y} \approx y$$

y hat and **y** are vectors

Last activation function: **None** (identity)

Loss: MSE, also averaged over the elements of the label vector

$$J(\Theta) = \frac{1}{2mk} \sum_{j=1}^{m} ||\hat{y}^{(j)} - y^{(j)}||_2^2 = \boxed{\frac{1}{2mk} \sum_{j=1}^{m} \sum_{i=1}^{k} (\hat{y}_i^{(j)} - y_i^{(j)})^2}$$

Last week - Multi-class **classification**

$$h(x) = g_2(W_2\, g_1(W_1 x + b_1) + b_2) = \hat{y} \approx y$$

y hat and **y** are vectors

Last activation function: Softmax

Loss: Cross-entropy

$$J(\theta) = -\frac{1}{m}\sum_{j=1}^{m}\sum_{i=1}^{k} y_i log(\hat{y}_i)$$

one-hot

$$\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \qquad \begin{bmatrix} 0.64 \\ 0.21 \\ 0.06 \\ 0.09 \end{bmatrix}$$

Last week - Multi-class **classification**

$$h(x) = g_2(W_2\ g_1(W_1 x + b_1) + b_2) = \hat{y} \approx y$$

Last activation function: **Softmax**

In PyTorch, already part of CE loss, no need to explicitly add it to our network.

Loss: Cross-entropy

$$J(\theta) = -\frac{1}{m}\sum_{j=1}^{m}\sum_{i=1}^{k} y_i log(\hat{y}_i)$$

$$\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \qquad \begin{bmatrix} 0.64 \\ 0.21 \\ 0.06 \\ 0.09 \end{bmatrix}$$

Training an MLP

We will use gradient descent…

```
repeat until convergence   {
    for $\forall\, \theta \in \boldsymbol{\Theta}$   {
        $grad_\theta = \frac{\partial}{\partial\theta} J(\boldsymbol{\Theta})$
    }
    for $\forall\, \theta \in \boldsymbol{\Theta}$   {
        $\theta = \theta - \alpha\, grad_\theta$
    }
}
```

Training an MLP

We will use gradient descent…

```
repeat until convergence {
    for $\forall\, \theta \in \Theta$ {
        $grad_\theta = \frac{\partial}{\partial \theta} J(\Theta)$
    }
    for $\forall\, \theta \in \Theta$ {
        $\theta = \theta - \alpha\, grad_\theta$
    }
}
```

We need to calculate the derivative of the loss function w.r.t. each parameter. This gives us the gradient vector, which we use to update the parameters.

Let's expand the matrix form of the hypothesis function to make things easier!

Multilayer Perceptron (MLP) expanded

The hypothesis function of a two-layer MLP neural network:

$$h(x) = g_2(W_2\ g_1(W_1 x + b_1) + b_2) = \hat{y} \approx y$$

Expanded:

$$\begin{aligned} h(x) = w_{2,1,1}\, g(&x_1 w_{1,1,1} + x_2 w_{1,1,2} + \cdots + x_n w_{1,1,n} + b_{1,1}) + \\ &w_{2,1,2}\, g(x_1 w_{1,2,1} + x_2 w_{1,2,2} + \cdots + x_n w_{1,2,n} + b_{1,2}) + b_{2,1} = \hat{y} \approx y \end{aligned}$$

Multilayer Perceptron (MLP) expanded

The hypothesis function of a two-layer MLP neural network:

$$h(x) = g_2(W_2\, g_1(W_1 x + b_1) + b_2) = \hat{y} \approx y$$

Expanded:

$$\begin{aligned} h(x) = {} & w_{2,1,1}\, g(x_1 w_{1,1,1} + x_2 w_{1,1,2} + \cdots + x_n w_{1,1,n} + b_{1,1}) + \\ & w_{2,1,2}\, g(x_1 w_{1,2,1} + x_2 w_{1,2,2} + \cdots + x_n w_{1,2,n} + b_{1,2}) + b_{2,1} = \hat{y} \approx y \end{aligned}$$

$$J(\Theta) = \frac{1}{2m} \sum_{j=1}^{m} (h(x^{(j)}) - y^{(j)})^2$$

$$\Theta = \{w_{1,1,1}, w_{1,1,2}, \ldots, b_{1,1}, \ldots\}$$

Multilayer Perceptron (MLP) expanded

We need to calculate the derivative of the loss function w.r.t. each parameter!

$$\begin{aligned} h(x) = w_{2,1,1}\, g(x_1 w_{1,1,1} + x_2 w_{1,1,2} + \cdots + x_n w_{1,1,n} + b_{1,1}) + \\ w_{2,1,2}\, g(x_1 w_{1,2,1} + x_2 w_{1,2,2} + \cdots + x_n w_{1,2,n} + b_{1,2}) + b_{2,1} = \hat{y} \approx y \end{aligned}$$

$$J(\Theta) = \frac{1}{2m} \sum_{j=1}^{m} (h(x^{(j)}) - y^{(j)})^2$$

$$\frac{\partial J(\Theta)}{\partial b_{2,1}} = \ ?$$

$$\frac{\partial J(\Theta)}{\partial w_{2,1,2}} = \ ?$$

$$\cdots$$

Multilayer Perceptron (MLP) expanded

How do we calculate the derivative of composite functions?

$$\frac{\partial J(\Theta)}{\partial b_{2,1}} = \ ?$$

$$\frac{\partial J(\Theta)}{\partial w_{2,1,2}} = \ ?$$

$$\cdots$$

Chain rule

Reminder: Derivatives of composite functions

$$(f \circ g)' = (f' \circ g) \cdot g'$$

The same with **Leibniz-notation**:

$$\frac{\partial u}{\partial x} = \frac{\partial u}{\partial z} \cdot \frac{\partial z}{\partial x}$$ ahol $$z := g(x),\ u := f(z)$$

Chain rule

Reminder: Derivatives of composite functions

$$(f \circ g)' = (f' \circ g) \cdot g'$$

The same with **Leibniz-notation**:

$$\frac{\partial u}{\partial x} = \frac{\partial u}{\partial z} \cdot \frac{\partial z}{\partial x} \quad \text{ahol} \qquad z := g(x),\ u := f(z)$$

The derivative of the expression **u** with respect to **x**

The rule for differentiating composite functions is also known as the **chain rule**.

Chain rule - An example

Example: Calculate the derivative of the sigmoid function!

$$f(x) = \frac{1}{1 + e^{-x}} \qquad \frac{\partial f(x)}{\partial x} = \ ?$$

Chain rule - An example

Example: Calculate the derivative of the sigmoid function!

$$f(x) = \frac{1}{1 + e^{-x}} \qquad \frac{\partial f(x)}{\partial x} = \ ?$$

Let's define intermediate variables for easier derivation!

$$z := -x$$

$$q := 1 + e^{z}$$

$$f(x) := \frac{1}{q} = q^{-1}$$

Chain rule - An example

Chain rule: We calculate the derivatives of the intermediate variables with respect to the chained variable, then multiply the results together.

$$\frac{\partial f(x)}{\partial x} = \frac{\partial f(x)}{\partial q} \cdot \frac{\partial q}{\partial z} \cdot \frac{\partial z}{\partial x}$$

$$z := -x$$

$$q := 1 + e^z$$

$$f(x) := \frac{1}{q} = q^{-1}$$

Chain rule - An example

Chain rule: We calculate the derivatives of the intermediate variables with respect to the chained variable, then multiply the results together.

$$\frac{\partial f(x)}{\partial x} = \frac{\partial f(x)}{\partial q} \cdot \frac{\partial q}{\partial z} \cdot \frac{\partial z}{\partial x}$$

$$z := -x$$

$$q := 1 + e^{z}$$

$$f(x) := \frac{1}{q} = q^{-1}$$

Reminder: A list of rules to calculate derivatives https://homepage.cs.uiowa.edu/~stroyan/CTLC3rdEd/3rdCTLCText/Chapters/ch6.pdf

Chain rule - An example

Chain rule: We calculate the derivatives of the intermediate variables with respect to the chained variable, then multiply the results together.

$$\frac{\partial f(x)}{\partial x} = \frac{\partial f(x)}{\partial q} \cdot \frac{\partial q}{\partial z} \cdot \frac{\partial z}{\partial x}$$

$$z := -x$$

$$q := 1 + e^{z}$$

$$f(x) := \frac{1}{q} = q^{-1}$$

$$\frac{\partial z}{\partial x} = -1$$

$$\frac{\partial q}{\partial z} = e^{z} = e^{-x}$$

$$\frac{\partial f(x)}{\partial q} = -q^{-2} = -\frac{1}{(1+e^{-x})^{2}}$$

Chain rule - An example

Chain rule: We calculate the derivatives of the intermediate variables with respect to the chained variable, then multiply the results together.

$$\frac{\partial f(x)}{\partial x} = \frac{\partial f(x)}{\partial q} \cdot \frac{\partial q}{\partial z} \cdot \frac{\partial z}{\partial x} = \boxed{(-1) \cdot (e^{-x}) \cdot -\frac{1}{(1+e^{-x})^2}}$$

$$z := -x$$

$$q := 1 + e^z$$

$$f(x) := \frac{1}{q} = q^{-1}$$

$$\frac{\partial z}{\partial x} = -1$$

$$\frac{\partial q}{\partial z} = e^z = e^{-x}$$

$$\frac{\partial f(x)}{\partial q} = -q^{-2} = -\frac{1}{(1+e^{-x})^2}$$

Chain rule - An example

Example: Calculate the derivative of the sigmoid function!

$$f(x) = \frac{1}{1+e^{-x}} \qquad \frac{\partial f(x)}{\partial x} = \; ?$$

$$\frac{\partial f(x)}{\partial x} = \frac{\partial f(x)}{\partial q} \cdot \frac{\partial q}{\partial z} \cdot \frac{\partial z}{\partial x} = \boxed{(-1) \cdot (e^{-x}) \cdot -\frac{1}{(1+e^{-x})^2}} = \frac{e^{-x}}{(1+e^{-x})^2}$$

Chain rule - An example

Example: Calculate the derivative of the sigmoid function!

$$f(x) = \frac{1}{1+e^{-x}} \qquad \frac{\partial f(x)}{\partial x} = \ ?$$

$$\frac{\partial f(x)}{\partial x} = \frac{\partial f(x)}{\partial q} \cdot \frac{\partial q}{\partial z} \cdot \frac{\partial z}{\partial x} = \boxed{(-1) \cdot (e^{-x}) \cdot -\frac{1}{(1+e^{-x})^2}} = \frac{e^{-x}}{(1+e^{-x})^2}$$

$$= \ \ldots \ = f(x)(1-f(x))$$

Training a Multilayer Perceptron (MLP)

Back to MLP…

How do we calculate the derivative of composite functions?

$$J(\Theta) = \frac{1}{2m}\sum_{j=1}^{m}(h(x^{(j)}) - y^{(j)})^2$$

$$\begin{aligned} h(x) = w_{2,1,1}g(x_1 w_{1,1,1} + x_2 w_{1,1,2} + \cdots + x_n w_{1,1,n} + b_{1,1}) + \\ w_{2,1,2}g(x_1 w_{1,2,1} + x_2 w_{1,2,2} + \cdots + x_n w_{1,2,n} + b_{1,2}) + b_{2,1} = \hat{y} \approx y \end{aligned}$$

$$\frac{\partial J(\Theta)}{\partial b_{2,1}} = \ ? \qquad \frac{\partial J(\Theta)}{\partial w_{2,1,1}} = \ ? \qquad \ldots$$

Training a Multilayer Perceptron (MLP)

Back to MLP…

How do we calculate the derivative of composite functions?

$$J(\Theta) = \frac{1}{2m} \sum_{j=1}^{m} (h(x^{(j)}) - y^{(j)})^2$$

$$\begin{aligned} h(x) = w_{2,1,1} g(x_1 w_{1,1,1} + x_2 w_{1,1,2} + \cdots + x_n w_{1,1,n} + b_{1,1}) + \\ w_{2,1,2} g(x_1 w_{1,2,1} + x_2 w_{1,2,2} + \cdots + x_n w_{1,2,n} + b_{1,2}) + b_{2,1} = \hat{y} \approx y \end{aligned}$$

$$\frac{\partial J(\Theta)}{\partial b_{2,1}} = \ ? \qquad \frac{\partial J(\Theta)}{\partial w_{2,1,1}} = \ ? \qquad \cdots$$

Let’s define intermediate variables!

Training a Multilayer Perceptron (MLP)

Let's define intermediate variables!

$$J(\Theta) = \frac{1}{2m}\sum_{j=1}^{m}(h(x^{(j)}) - y^{(j)})^2$$

$$\hat{y} := h(x)$$

$$J(\Theta) = \frac{1}{2m}\sum_{j=1}^{m}(\hat{y} - y)^2$$

Training a Multilayer Perceptron (MLP)

Let’s define intermediate variables!

$$\hat{y} = w_{2,1,1}\, g(x_1 w_{1,1,1} + x_2 w_{1,1,2} + \cdots + x_n w_{1,1,n} + b_{1,1}) +$$

$$w_{2,1,2}\, g(x_1 w_{1,2,1} + x_2 w_{1,2,2} + \cdots + x_n w_{1,2,n} + b_{1,2}) + b_{2,1}$$

$$z_1 := w_{2,1,1}\, g(x_1 w_{1,1,1} + x_2 w_{1,1,2} + \cdots + x_n w_{1,1,n} + b_{1,1})$$

$$z_2 := w_{2,1,2}\, g(x_1 w_{1,2,1} + x_2 w_{1,2,2} + \cdots + x_n w_{1,2,n} + b_{1,2})$$

$$\hat{y} = z_1 + z_2 + b_{2,1}$$

Training a Multilayer Perceptron (MLP)

Let’s define intermediate variables!

$$z_1 = w_{2,1,1}\,g(x_1 w_{1,1,1} + x_2 w_{1,1,2} + \cdots + x_n w_{1,1,n} + b_{1,1})$$

$$z_2 = w_{2,1,2}\,g(x_1 w_{1,2,1} + x_2 w_{1,2,2} + \cdots + x_n w_{1,2,n} + b_{1,2})$$

$$q_1 := x_1 w_{1,1,1} + x_2 w_{1,1,2} + \cdots + x_n w_{1,1,n} + b_{1,1}$$

$$q_2 := x_1 w_{1,2,1} + x_2 w_{1,2,2} + \cdots + x_n w_{1,2,n} + b_{1,2}$$

$$z_1 = w_{2,1,1}\,g(q_1)$$

$$z_2 = w_{2,1,2}\,g(q_2)$$

Training a Multilayer Perceptron (MLP)

We need gradients to use the gradient method!

$$\frac{\partial J(\Theta)}{\partial b_{2,1}} = \frac{\partial J(\Theta)}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial b_{2,1}}$$

$$J(\Theta) = \frac{1}{2m} \sum_{j=1}^{m} (\hat{y} - y)^2$$

$$\hat{y} = z_1 + z_2 + b_{2,1}$$

Training a Multilayer Perceptron (MLP)

We need gradients to use the gradient method!

$$\frac{\partial J(\Theta)}{\partial b_{2,1}} = \frac{\partial J(\Theta)}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial b_{2,1}}$$

$$J(\Theta) = \frac{1}{2m} \sum_{j=1}^{m} (\hat{y} - y)^2$$

$$\frac{\partial J(\Theta)}{\partial w_{2,1,1}} = \frac{\partial J(\Theta)}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_1} \cdot \frac{\partial z_1}{\partial w_{2,1,1}}$$

$$\hat{y} = z_1 + z_2 + b_{2,1}$$

$$z_1 = w_{2,1,1} g(x_1 w_{1,1,1} + x_2 w_{1,1,2} + \cdots + x_n w_{1,1,n} + b_{1,1})$$

Training a Multilayer Perceptron (MLP)

The derivative of the loss must be calculated w.r.t. each parameter!

$$\frac{\partial J(\Theta)}{\partial b_{2,1}} = \frac{\partial J(\Theta)}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial b_{2,1}}$$

$$\frac{\partial J(\Theta)}{\partial w_{2,1,1}} = \frac{\partial J(\Theta)}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_1} \cdot \frac{\partial z_1}{\partial w_{2,1,1}}$$

$$\frac{\partial J(\Theta)}{\partial w_{1,1,1}} = \frac{\partial J(\Theta)}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_1} \cdot \frac{\partial z_1}{\partial q_1} \cdot \frac{\partial q_1}{\partial w_{1,1,1}}$$

$$\cdots$$

Training a Multilayer Perceptron (MLP)

The derivative of the loss must be calculated w.r.t. each parameter!

$$\frac{\partial J(\Theta)}{\partial b_{2,1}} = \frac{\partial J(\Theta)}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial b_{2,1}}$$

$$\frac{\partial J(\Theta)}{\partial w_{2,1,1}} = \frac{\partial J(\Theta)}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_1} \cdot \frac{\partial z_1}{\partial w_{2,1,1}}$$

$$\frac{\partial J(\Theta)}{\partial w_{1,1,1}} = \frac{\partial J(\Theta)}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_1} \cdot \frac{\partial z_1}{\partial q_1} \cdot \frac{\partial q_1}{\partial w_{1,1,1}}$$

$$\cdots$$

It seems like a lot of computation, calculating all expressions with respect to all parameters.

Can it be done more efficiently?

Training a Multilayer Perceptron (MLP)

The derivative of the loss must be calculated w.r.t. each parameter!

$$\frac{\partial J(\Theta)}{\partial b_{2,1}} = \boxed{\frac{\partial J(\Theta)}{\partial \hat{y}}} \cdot \frac{\partial \hat{y}}{\partial b_{2,1}}$$

$$\frac{\partial J(\Theta)}{\partial w_{2,1,1}} = \boxed{\frac{\partial J(\Theta)}{\partial \hat{y}}} \cdot \boxed{\frac{\partial \hat{y}}{\partial z_1}} \cdot \frac{\partial z_1}{\partial w_{2,1,1}}$$

$$\frac{\partial J(\Theta)}{\partial w_{1,1,1}} = \boxed{\frac{\partial J(\Theta)}{\partial \hat{y}}} \cdot \boxed{\frac{\partial \hat{y}}{\partial z_1}} \cdot \frac{\partial z_1}{\partial q_1} \cdot \frac{\partial q_1}{\partial w_{1,1,1}}$$

$$\cdots$$

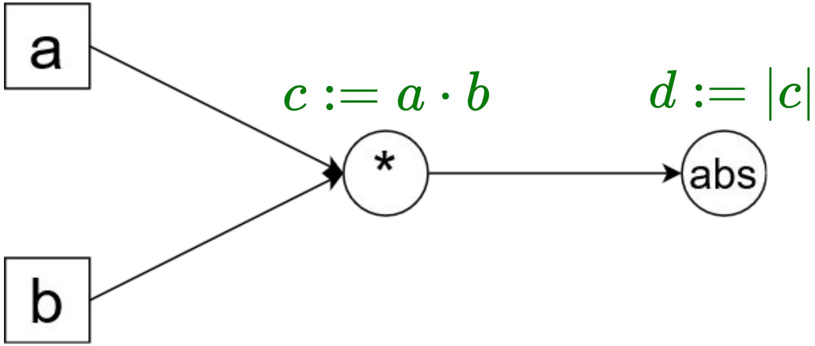
The same members appear multiple times in the derivatives. **It is useless to calculate them multiple times.**

Gradient method + keeping track of sub-results
→ **Backpropagation algorithm**

Backpropagation algorithm

Backpropagation algorithm

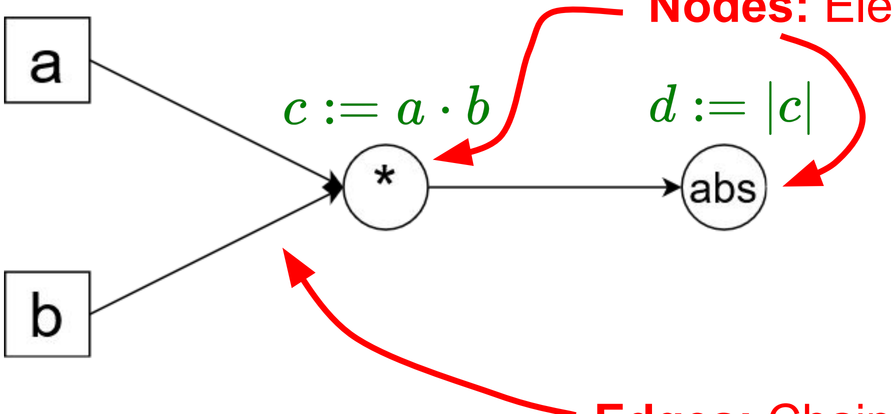
Computational graph: Representing complicated expressions



Backpropagation algorithm

Backpropagation algorithm

Computational graph: Representing complicated expressions



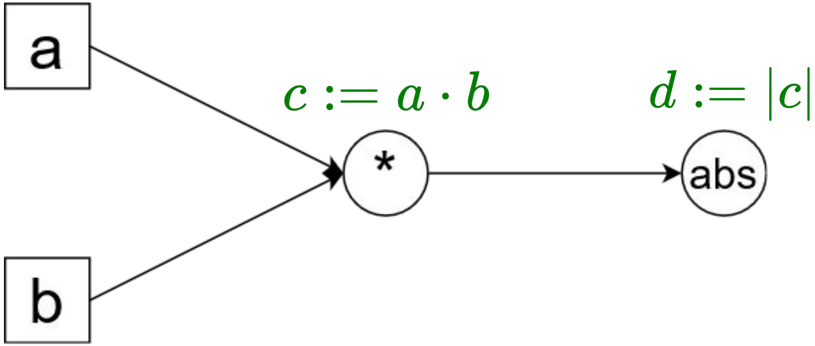
Nodes: Elementary operations

Edges: Chaining elementary operations together

Backpropagation algorithm

Backpropagation algorithm

Computational graph: Representing complicated expressions

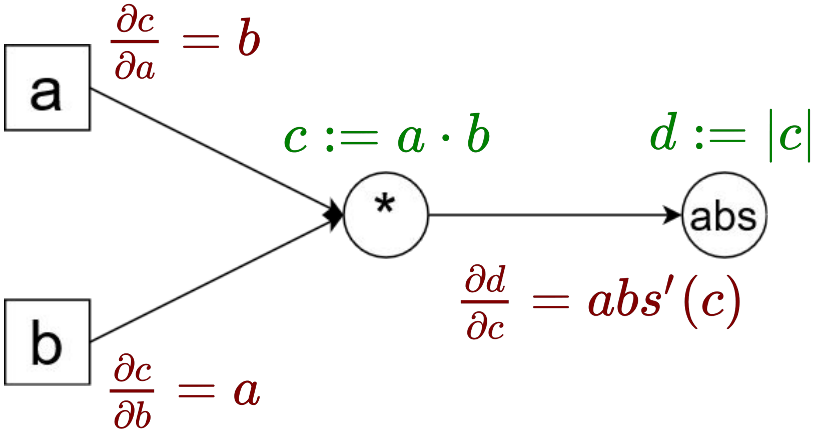


The graph tells us **what other intermediate variable need to be evaluated** in order to evaluate a given variable.

Backpropagation algorithm

Backpropagation algorithm

Computational graph: Representing complicated expressions



$\frac{\partial c}{\partial a} = b$

$c := a \cdot b$

$d := |c|$

$\frac{\partial d}{\partial c} = abs'(c)$

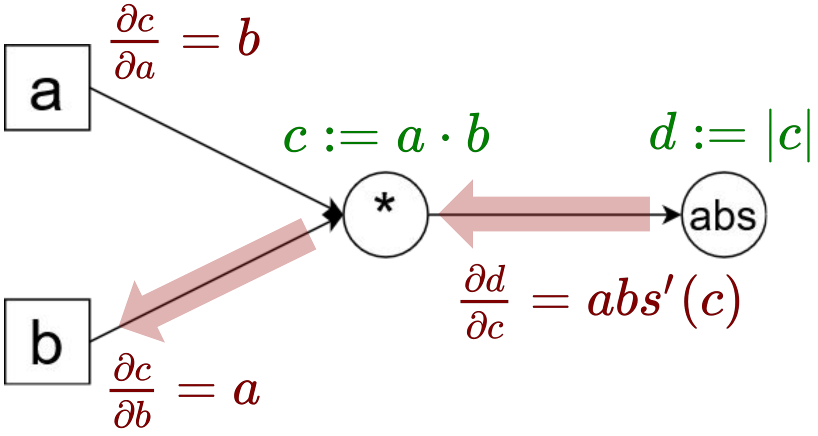
$\frac{\partial c}{\partial b} = a$

We only need to calculate **the derivative of elementary operations.**

Backpropagation algorithm

Backpropagation algorithm

Computational graph: Representing complicated expressions


a
b
*
abs
$\frac{\partial c}{\partial a} = b$
$c := a \cdot b$
$d := |c|$
$\frac{\partial d}{\partial c} = abs'(c)$
$\frac{\partial c}{\partial b} = a$

$$\frac{\partial d}{\partial b} = \frac{\partial d}{\partial c} \cdot \frac{\partial c}{\partial b}$$

Then, following the chain rule, take the product of **the elementary derivatives** along the corresponding edges.

Backpropagation algorithm

Backpropagation algorithm

By linking elementary operations and defining intermediate variables, we construct a **computational graph**.

The computational graph specifies which other variables need to be evaluated in order to evaluate a given variable.

Backpropagation algorithm

Backpropagation algorithm

By linking elementary operations and defining intermediate variables, we construct a **computational graph**.

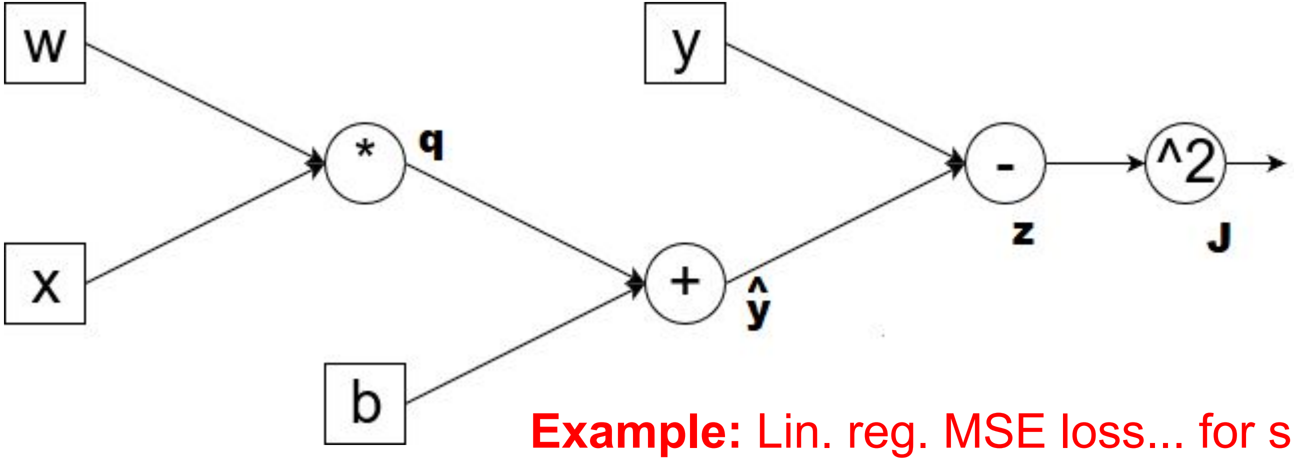
The computational graph specifies which other variables need to be evaluated in order to evaluate a given variable.

- We only calculate the **derivative of elementary operations**.
- We take **the product** of the elementary derivatives along the corresponding edges, following the chain rule.

Backpropagation algorithm

An example for a computational graph

$$J = ((wx + b) - y)^2$$



$$q = wx$$

$$\hat{y} = q + b$$

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

$$J = z^2$$

Example: Lin. reg. MSE loss... for simplicity's sake, we have one variable, we omit the multiplication by ½, and our sample consists of a single element.

Backpropagation algorithm

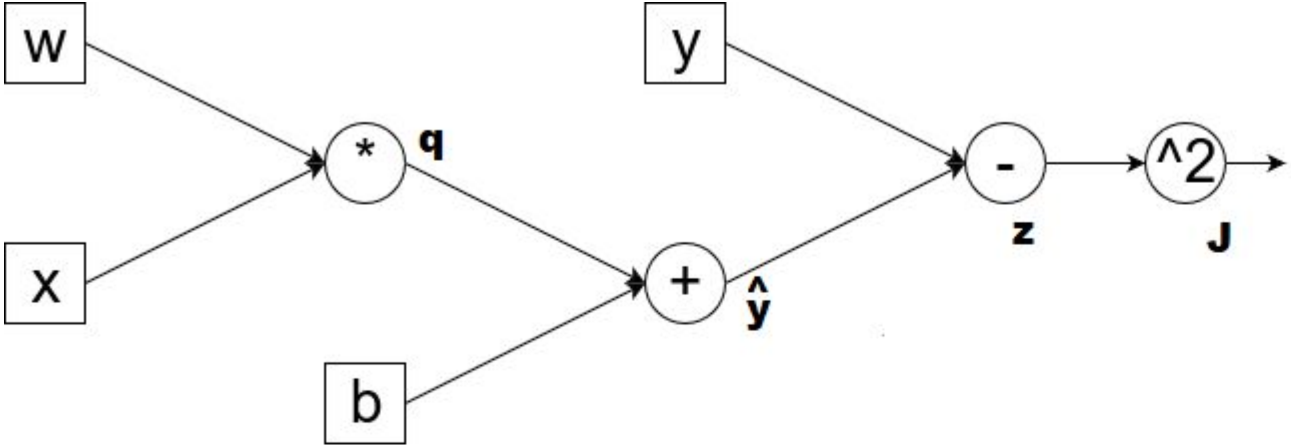
Step 1: Symbolic derivation

$$q = wx$$

$$\hat{y} = q + b$$

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

$$J = z^2$$

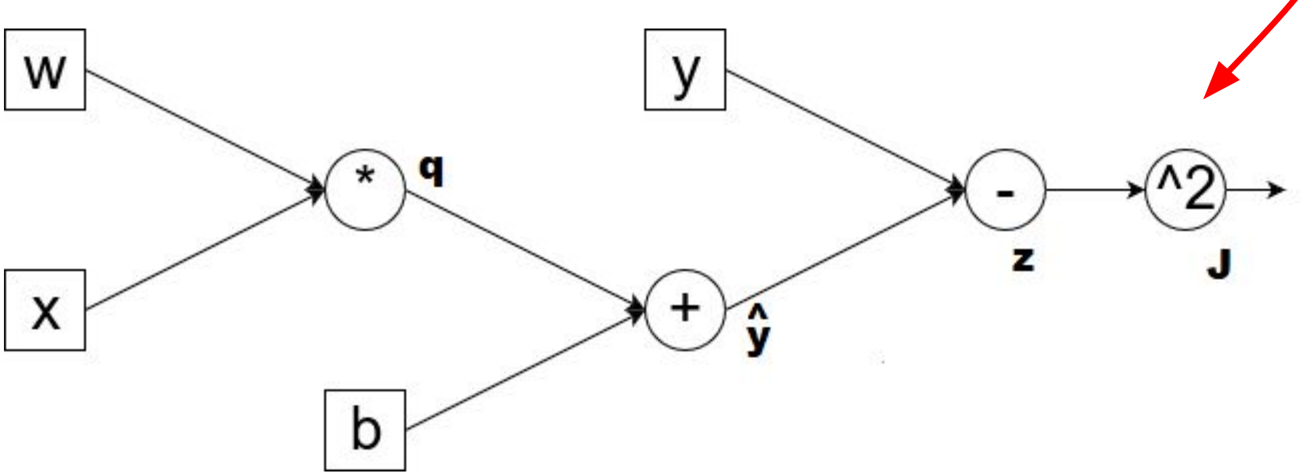


$$\frac{\partial J}{\partial b} = \ ? \qquad \frac{\partial J}{\partial w} = \ ?$$

(**Question:** Is it always a tree graph?)

Backpropagation algorithm

Step 1: Symbolic derivation



$$q = wx$$

$$\hat{y} = q + b$$

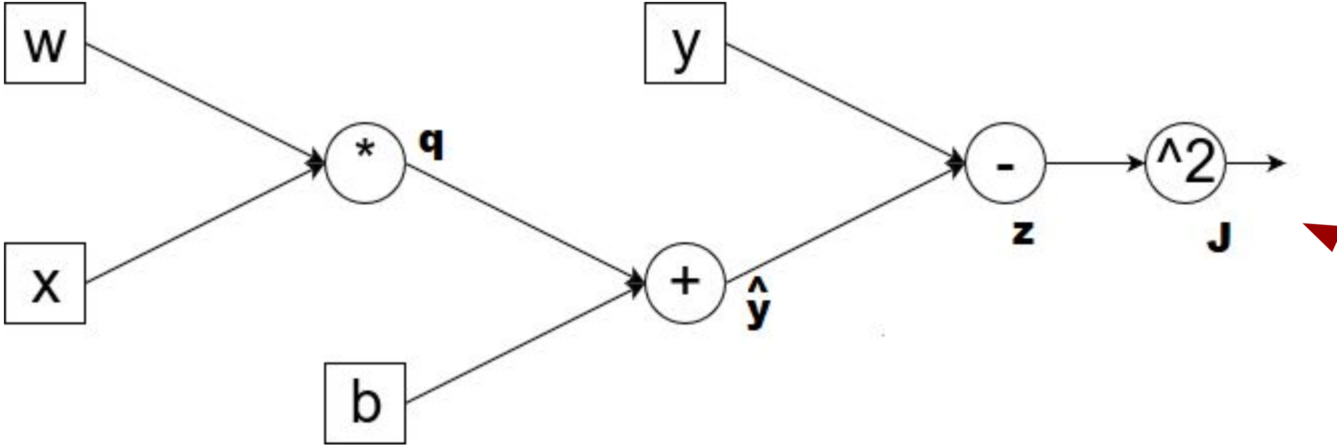
$$z = y - \hat{y}$$

$$J = z^2$$

$$\frac{\partial J}{\partial b} = ? \qquad \frac{\partial J}{\partial w} = ?$$

Backpropagation algorithm

Step 1: Symbolic derivation



$$q = wx$$

$$\hat{y} = q + b$$

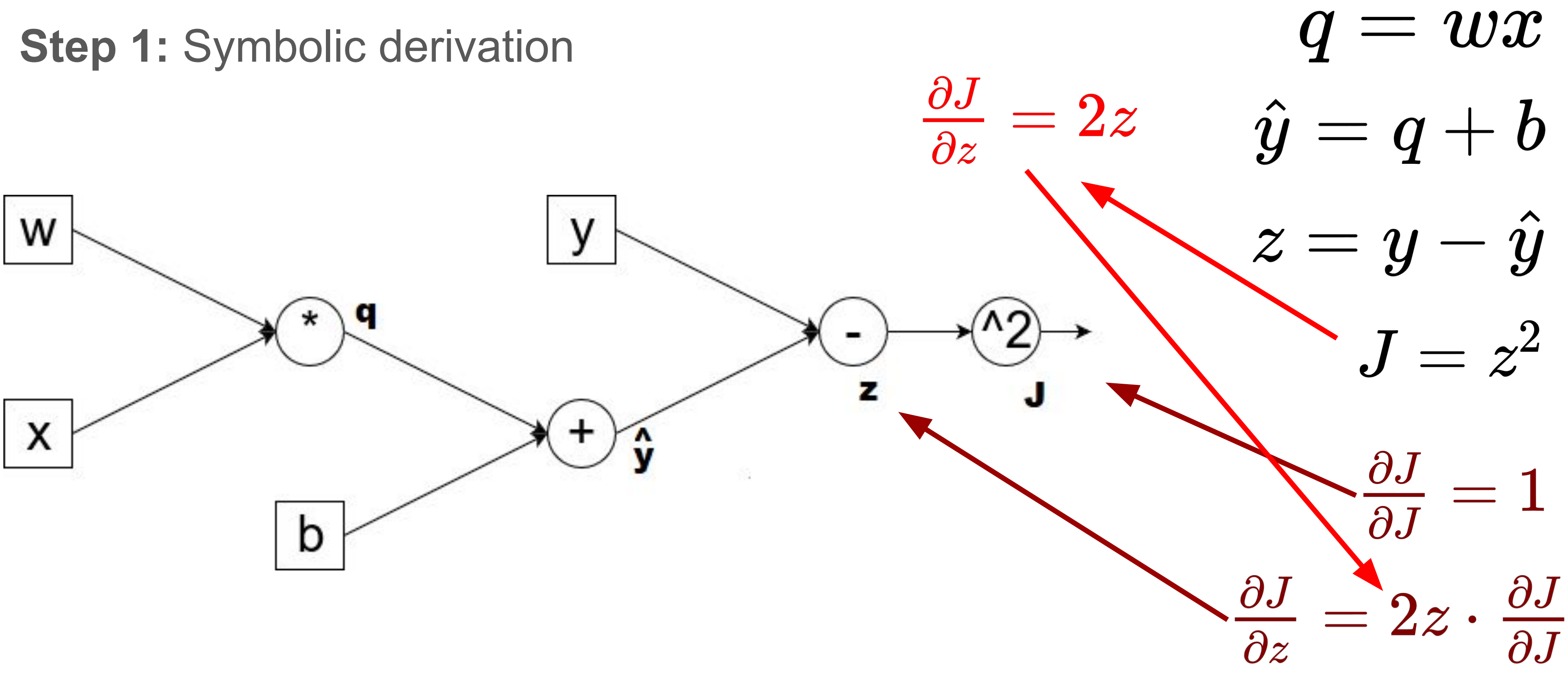
$$z = y - \hat{y}$$

$$J = z^2$$

$$\frac{\partial J}{\partial J} = 1$$

Backpropagation algorithm

Step 1: Symbolic derivation



$$q = wx$$

$$\hat{y} = q + b$$

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

$$J = z^2$$

$$\frac{\partial J}{\partial z} = 2z$$

$$\frac{\partial J}{\partial J} = 1$$

$$\frac{\partial J}{\partial z} = 2z \cdot \frac{\partial J}{\partial J}$$

Backpropagation algorithm

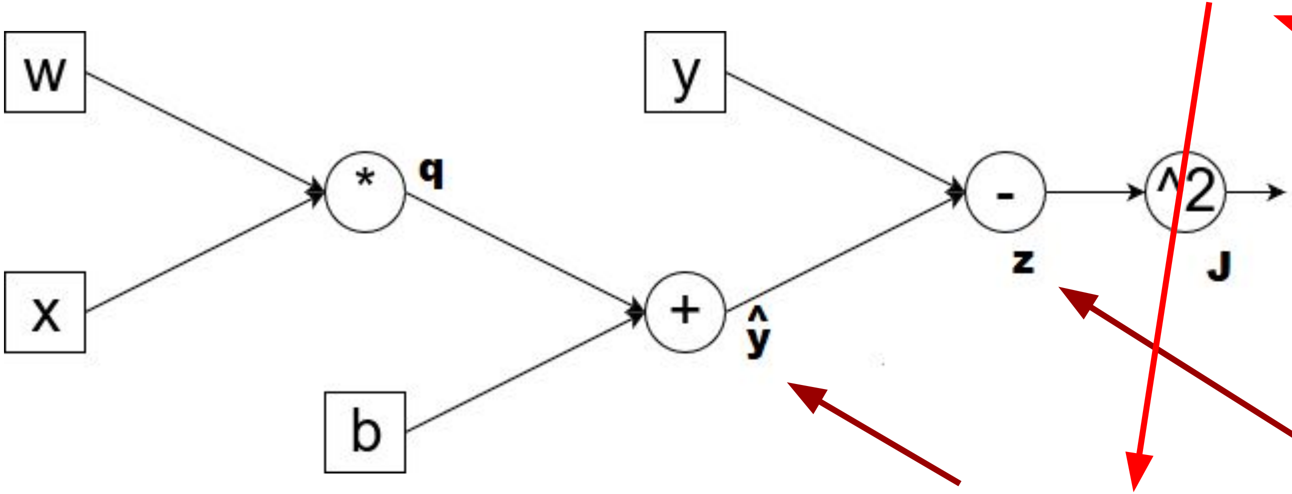
Step 1: Symbolic derivation

$$q = wx$$

$$\hat{y} = q + b$$

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

$$J = z^2$$



$$\frac{\partial z}{\partial \hat{y}} = -1$$

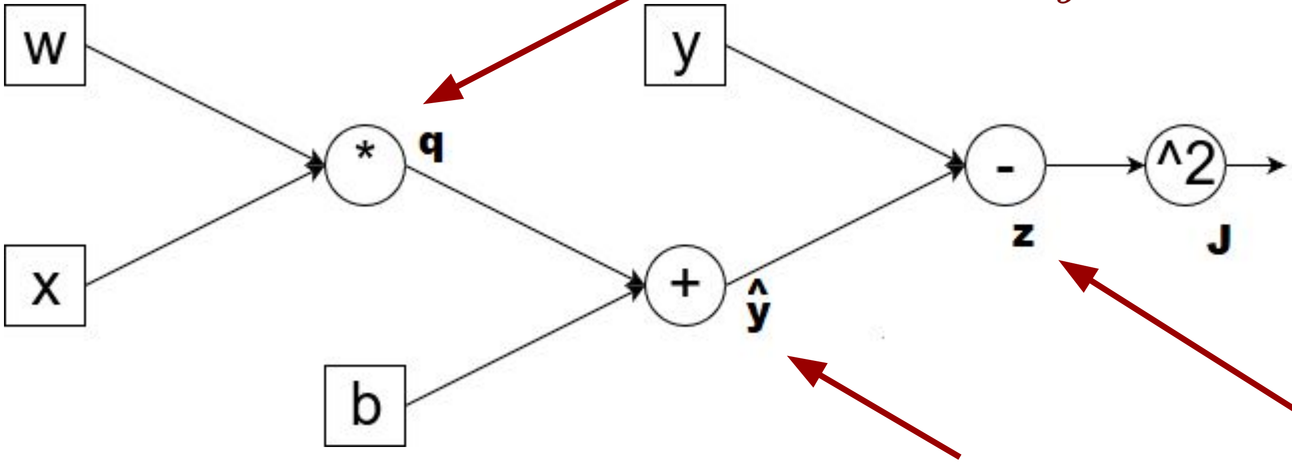
$$\frac{\partial J}{\partial J} = 1$$

$$\frac{\partial J}{\partial z} = 2z \cdot \frac{\partial J}{\partial J}$$

$$\frac{\partial J}{\partial \hat{y}} = -1 \cdot \frac{\partial J}{\partial z}$$

Backpropagation algorithm

Step 1: Symbolic derivation



$$q = wx$$

$$\hat{y} = q + b$$

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

$$J = z^2$$

$$\frac{\partial J}{\partial J} = 1$$

$$\frac{\partial J}{\partial z} = 2z \cdot \frac{\partial J}{\partial J}$$

$$\frac{\partial J}{\partial \hat{y}} = -1 \cdot \frac{\partial J}{\partial z}$$

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

$$\frac{\partial J}{\partial q} = 1 \cdot \frac{\partial J}{\partial \hat{y}}$$

Backpropagation algorithm

Step 1: Symbolic derivation

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

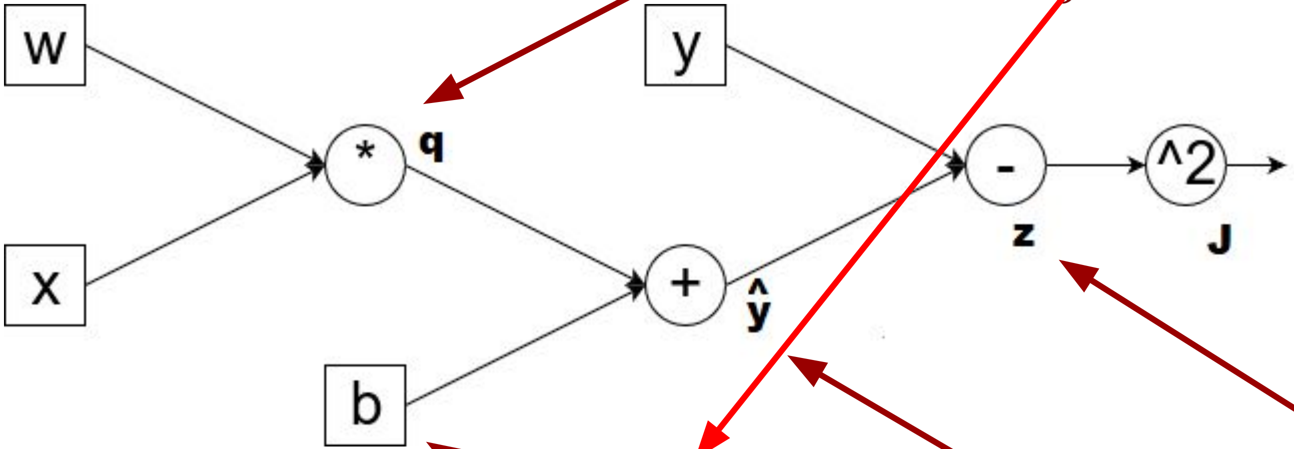
$$q = wx$$

$$\hat{y} = q + b$$

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

$$J = z^2$$

$$\frac{\partial J}{\partial q} = 1 \cdot \frac{\partial J}{\partial \hat{y}}$$



$$\frac{\partial J}{\partial J} = 1$$

$$\frac{\partial J}{\partial z} = 2z \cdot \frac{\partial J}{\partial J}$$

$$\frac{\partial J}{\partial \hat{y}} = -1 \cdot \frac{\partial J}{\partial z}$$

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

Backpropagation algorithm

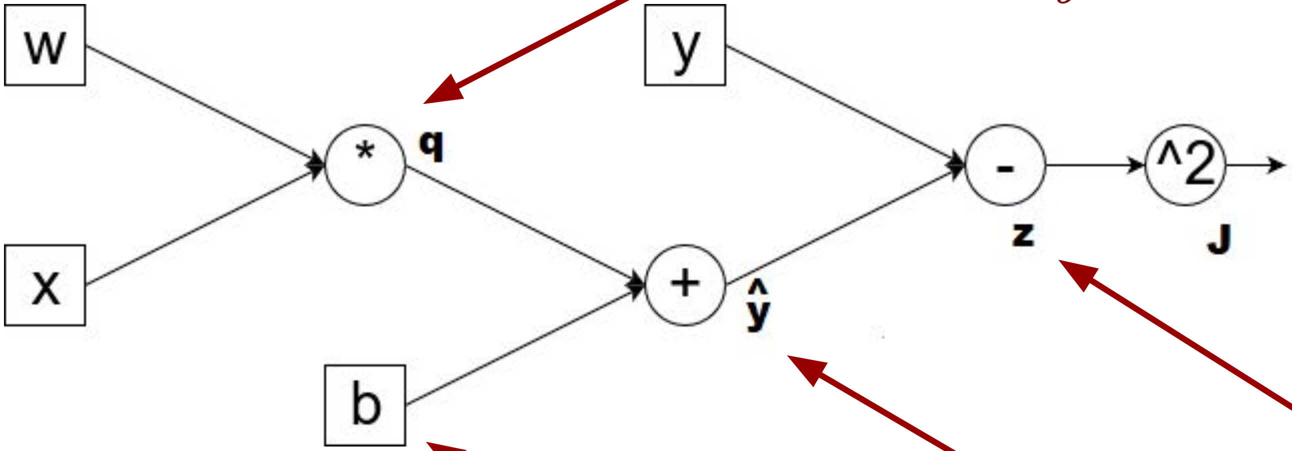
Step 1: Symbolic derivation

$$q = wx$$

$$\hat{y} = q + b$$

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

$$J = z^2$$



$$\frac{\partial q}{\partial w} = x$$

$$\frac{\partial J}{\partial w} = x \cdot \frac{\partial J}{\partial q}$$

$$\frac{\partial J}{\partial q} = 1 \cdot \frac{\partial J}{\partial \hat{y}}$$

$$\frac{\partial J}{\partial J} = 1$$

$$\frac{\partial J}{\partial z} = 2z \cdot \frac{\partial J}{\partial J}$$

$$\frac{\partial J}{\partial \hat{y}} = -1 \cdot \frac{\partial J}{\partial z}$$

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

Backpropagation algorithm

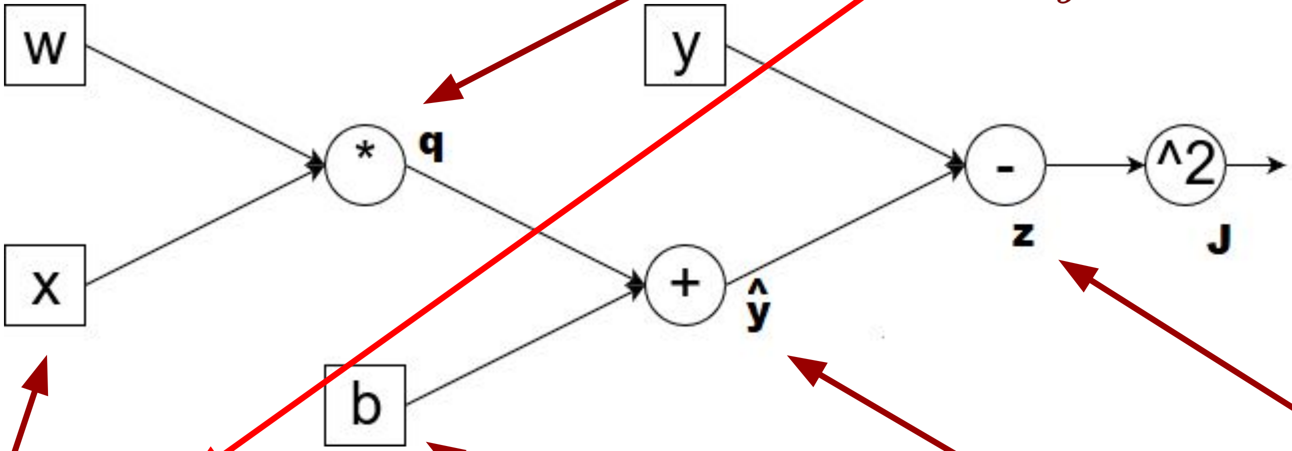
Step 1: Symbolic derivation

$$q = wx$$

$$\hat{y} = q + b$$

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

$$J = z^2$$



$$\frac{\partial q}{\partial x} = w$$

$$\frac{\partial J}{\partial w} = x \cdot \frac{\partial J}{\partial q}$$

$$\frac{\partial J}{\partial q} = 1 \cdot \frac{\partial J}{\partial \hat{y}}$$

$$\frac{\partial J}{\partial J} = 1$$

$$\frac{\partial J}{\partial z} = 2z \cdot \frac{\partial J}{\partial J}$$

$$\frac{\partial J}{\partial \hat{y}} = -1 \cdot \frac{\partial J}{\partial z}$$

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

$$\frac{\partial J}{\partial x} = w \cdot \frac{\partial J}{\partial q}$$

Backpropagation algorithm

Step 2: Parameter initialization

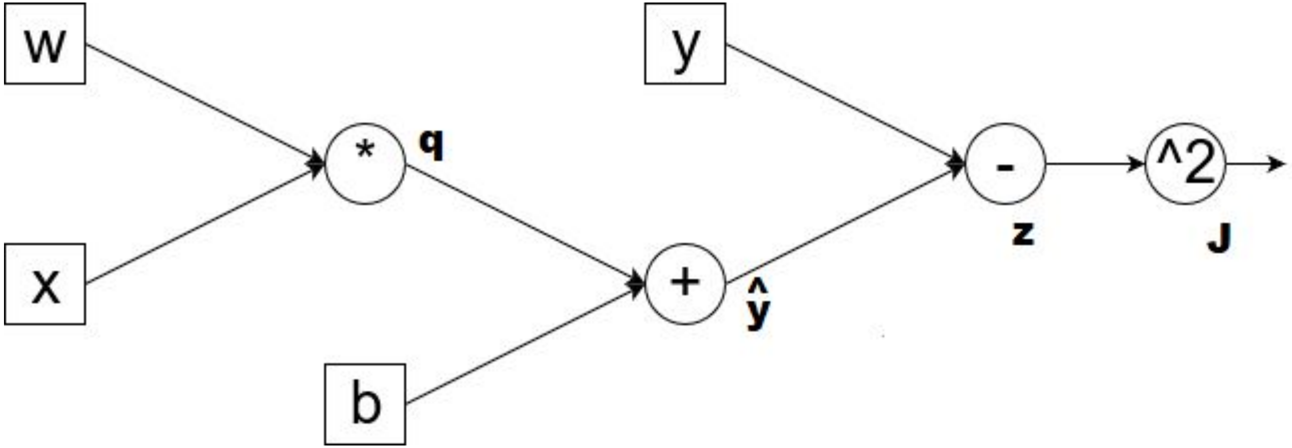
E.g., $w = 3,\ b = 4$

$$q = wx$$

$$\hat{y} = q + b$$

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

$$J = z^2$$



Backpropagation algorithm

Select the **initial value of parameters**, for example, randomly.

Step 2: Parameter initialization

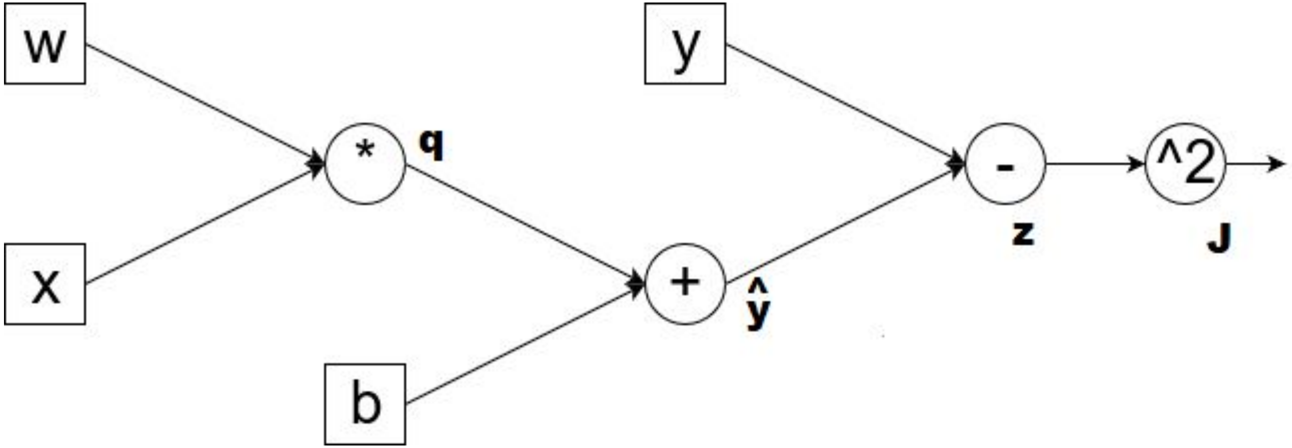
E.g., $w = 3,\ b = 4$

$$q = wx$$

$$\hat{y} = q + b$$

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

$$J = z^2$$



Backpropagation algorithm

Step 3: Choosing an example from the training set

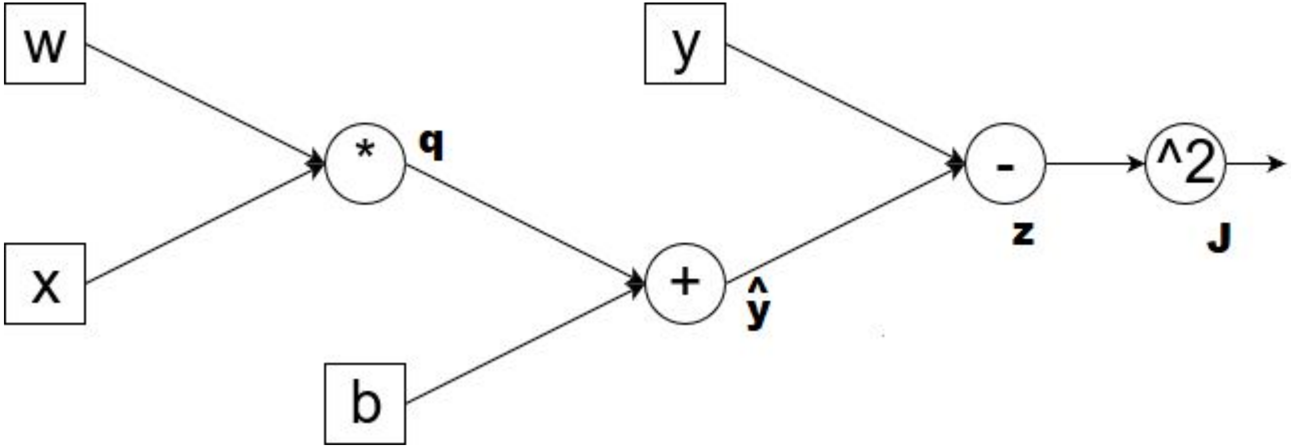
E.g., $x = 2,\ y = 5$

$$q = wx$$

$$\hat{y} = q + b$$

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

$$J = z^2$$



Backpropagation algorithm

In practice (SGD algorithm and variants) **a mini-batch of data is taken**, not necessarily a single example.

Step 3: Choosing an example from the training set

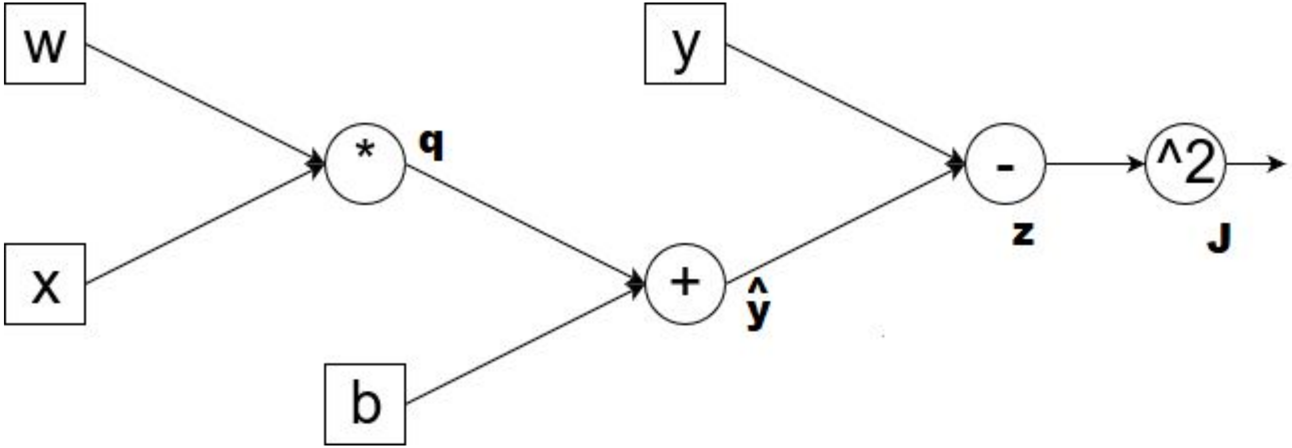
E.g., $x = 2,\ y = 5$

$$q = wx$$

$$\hat{y} = q + b$$

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

$$J = z^2$$

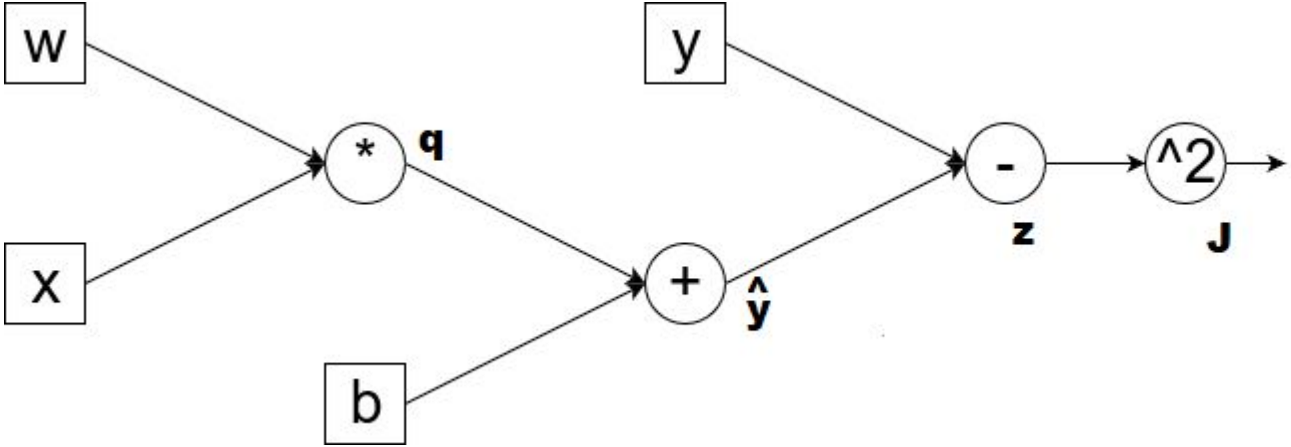


Backpropagation algorithm

Step 4: Substitution - feed forward

E.g., $x = 2,\ y = 5$ and $w = 3,\ b = 4$

$$q = wx$$
$$\hat{y} = q + b$$
$$z = y - \hat{y}$$
$$J = z^2$$



Backpropagation algorithm

We **substitute the input / output variables** with real data from the training set **and the parameter variables** with the current value of the parameters.

Step 4: Substitution - feed forward

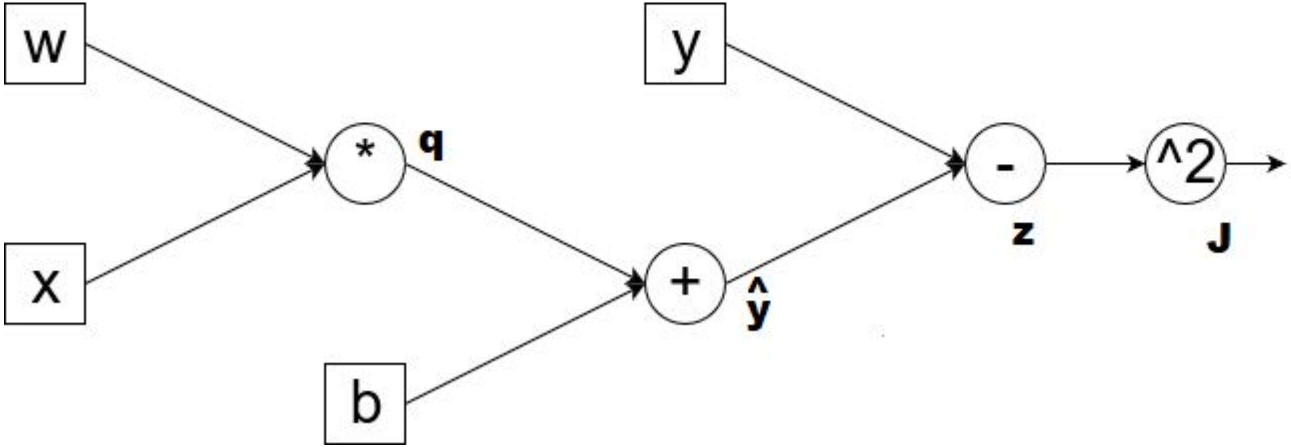
E.g., $x = 2,\ y = 5$ and $w = 3,\ b = 4$

$$q = wx$$

$$\hat{y} = q + b$$

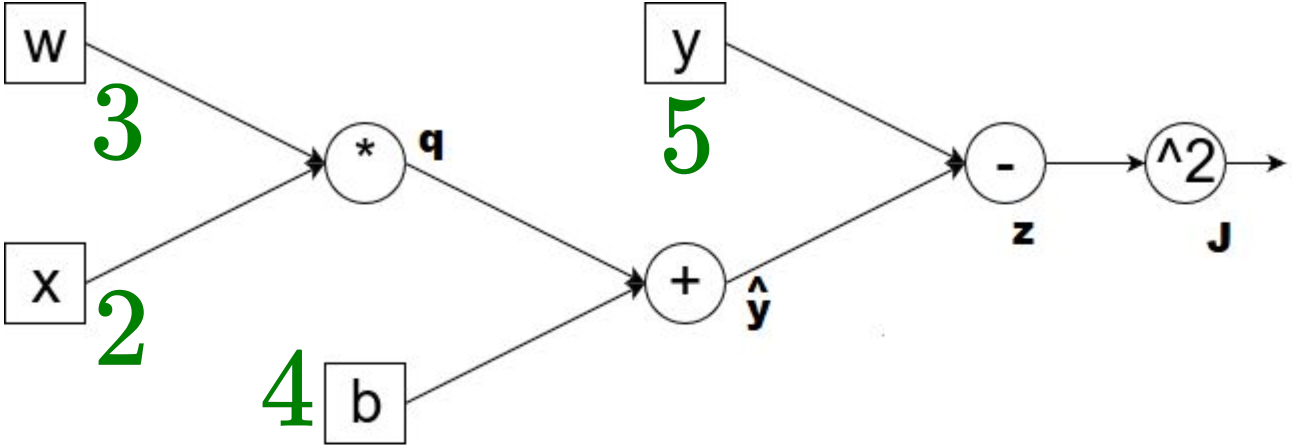
$$z = y - \hat{y}$$

$$J = z^2$$



Backpropagation algorithm

Step 4: Substitution - feed forward



$$q = wx$$

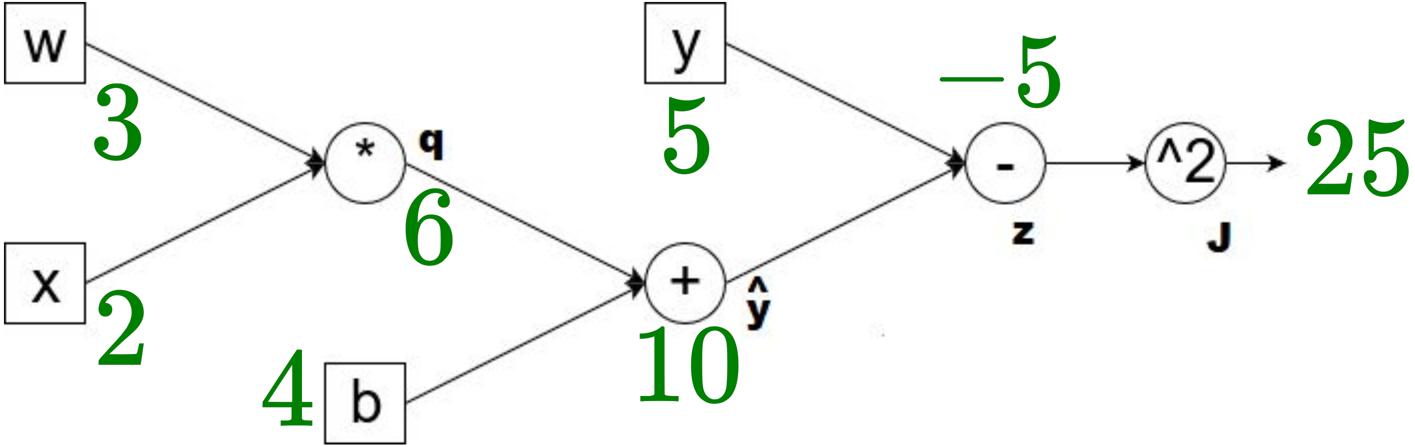
$$\hat{y} = q + b$$

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

$$J = z^2$$

Backpropagation algorithm

Step 4: Substitution - feed forward



$$q = wx$$

$$\hat{y} = q + b$$

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

$$J = z^2$$

Backpropagation algorithm

Step 5: Substitution - backpropagation

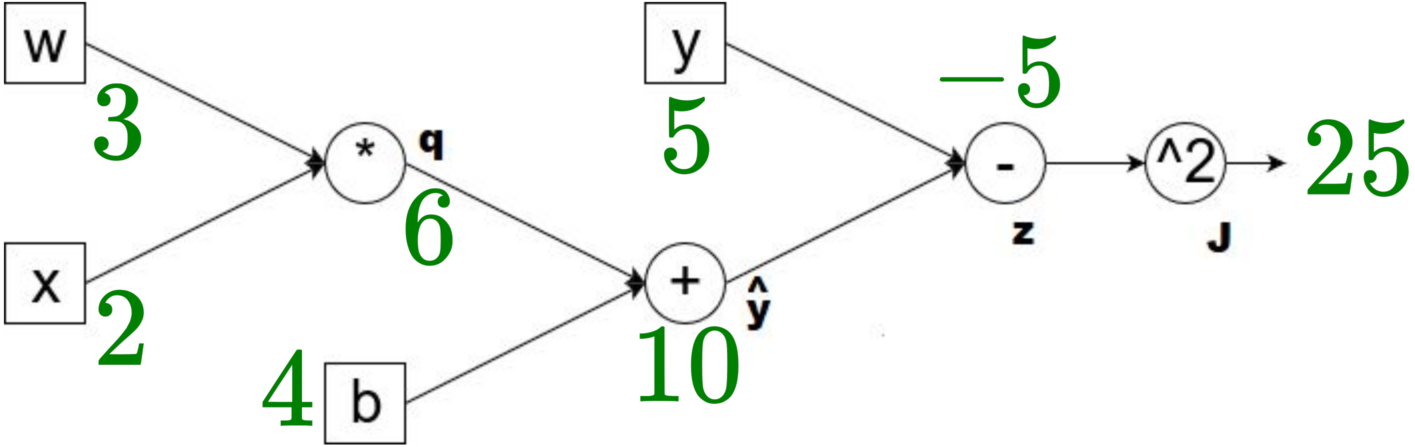
We have already calculated the gradients symbolically, now let's substitute them in!

$$q = wx$$

$$\hat{y} = q + b$$

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

$$J = z^2$$



Backpropagation algorithm

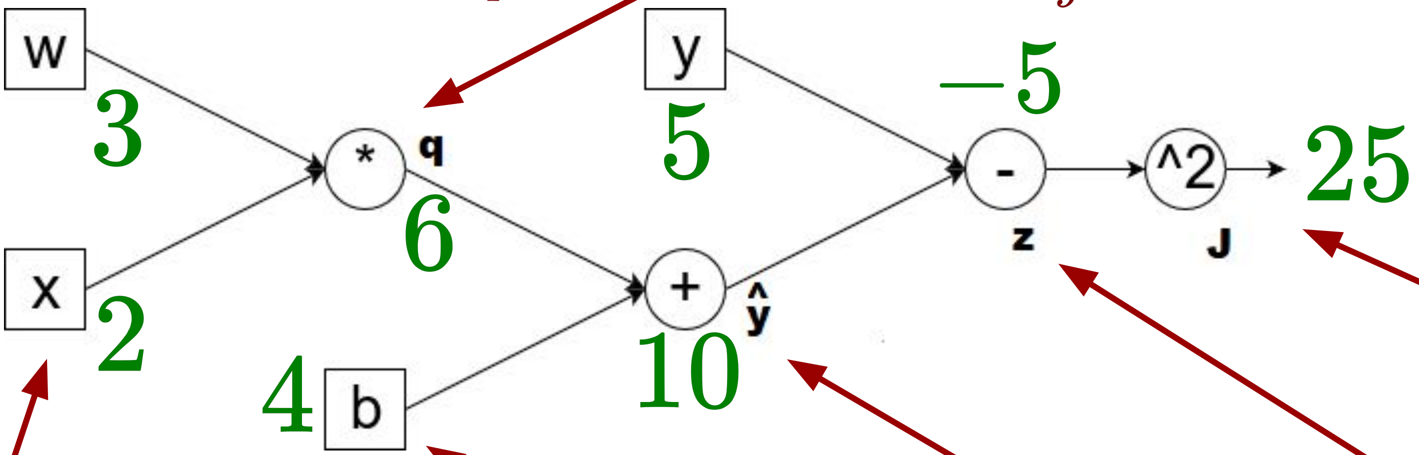
Step 5: Substitution - backpropagation

$$q = wx$$

$$\hat{y} = q + b$$

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

$$J = z^2$$



$$\frac{\partial J}{\partial w} = x \cdot \frac{\partial J}{\partial q}$$

$$\frac{\partial J}{\partial q} = 1 \cdot \frac{\partial J}{\partial \hat{y}}$$

$$\frac{\partial J}{\partial J} = 1$$

$$\frac{\partial J}{\partial x} = w \cdot \frac{\partial J}{\partial q}$$

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

$$\frac{\partial J}{\partial \hat{y}} = -1 \cdot \frac{\partial J}{\partial z}$$

$$\frac{\partial J}{\partial z} = 2z \cdot \frac{\partial J}{\partial J}$$

Backpropagation algorithm

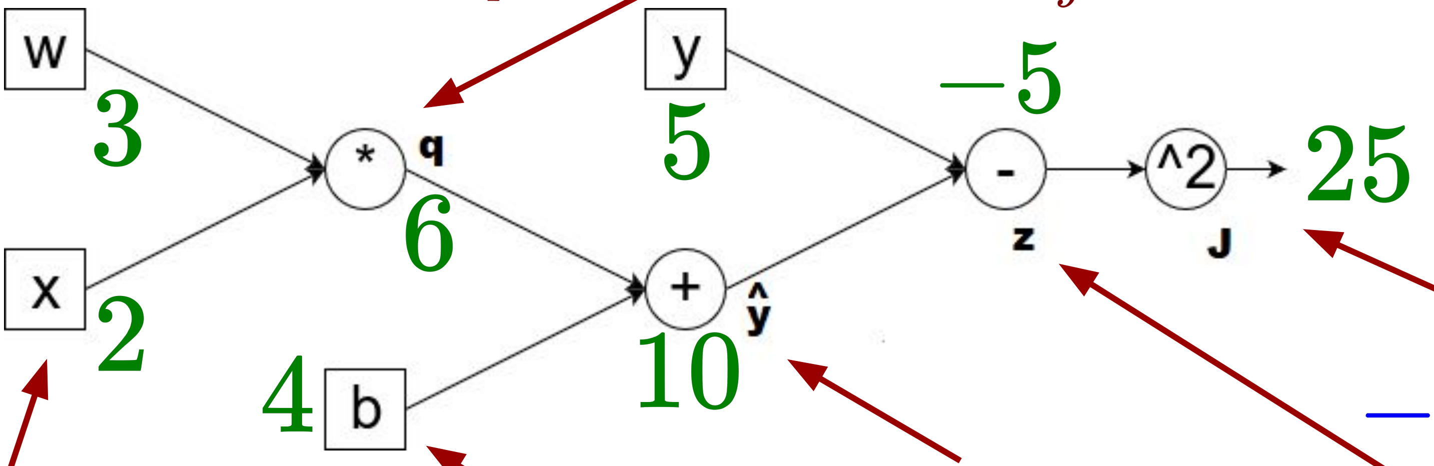
Step 5: Substitution - backpropagation

$$q = wx$$

$$\hat{y} = q + b$$

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

$$J = z^2$$



$$\frac{\partial J}{\partial w} = x \cdot \frac{\partial J}{\partial q}$$

$$\frac{\partial J}{\partial q} = 1 \cdot \frac{\partial J}{\partial \hat{y}}$$

$$1 \quad \frac{\partial J}{\partial J} = 1$$

$$-10 \quad \frac{\partial J}{\partial z} = 2z \cdot \frac{\partial J}{\partial J}$$

$$\frac{\partial J}{\partial \hat{y}} = -1 \cdot \frac{\partial J}{\partial z}$$

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

$$\frac{\partial J}{\partial x} = w \cdot \frac{\partial J}{\partial q}$$

Backpropagation algorithm

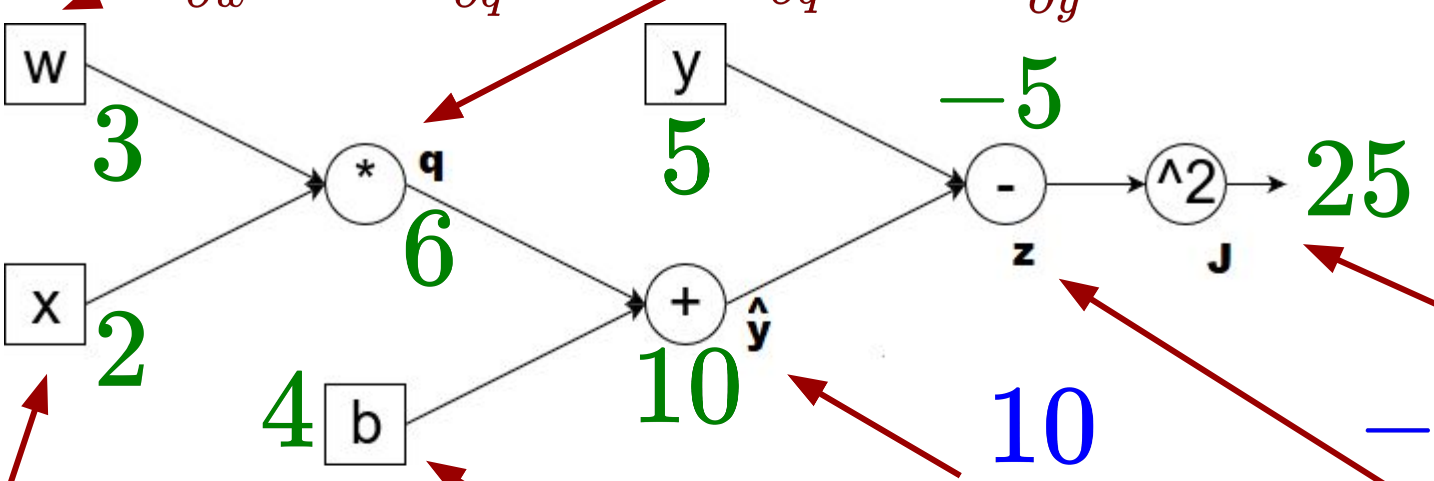
Step 5: Substitution - backpropagation

$$q = wx$$

$$\hat{y} = q + b$$

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

$$J = z^2$$



$$\frac{\partial J}{\partial w} = x \cdot \frac{\partial J}{\partial q}$$

$$\frac{\partial J}{\partial q} = 1 \cdot \frac{\partial J}{\partial \hat{y}}$$

$$\frac{\partial J}{\partial J} = 1$$

$$\frac{\partial J}{\partial x} = w \cdot \frac{\partial J}{\partial q}$$

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

$$\frac{\partial J}{\partial \hat{y}} = -1 \cdot \frac{\partial J}{\partial z}$$

$$\frac{\partial J}{\partial z} = 2z \cdot \frac{\partial J}{\partial J}$$

Backpropagation algorithm

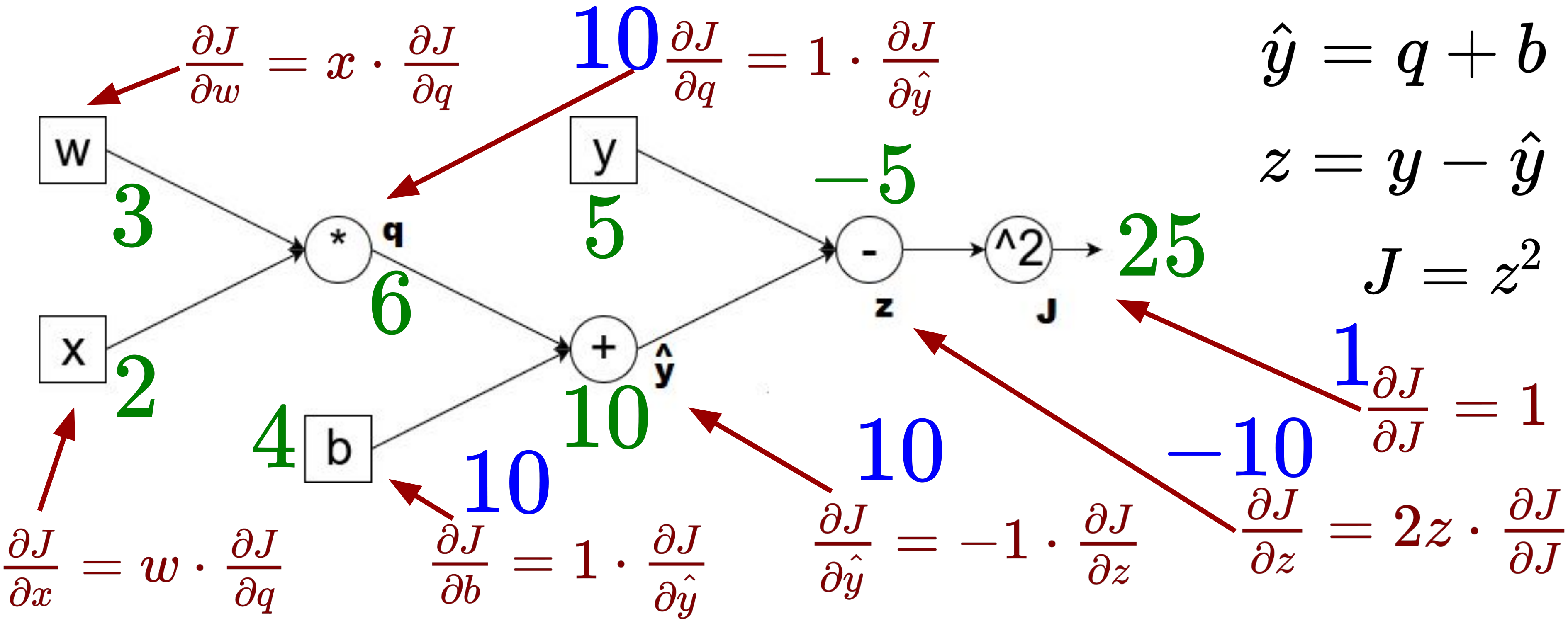
Step 5: Substitution - backpropagation

$$q = wx$$

$$\hat{y} = q + b$$

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

$$J = z^2$$



$$\frac{\partial J}{\partial w} = x \cdot \frac{\partial J}{\partial q}$$

$$\frac{\partial J}{\partial q} = 1 \cdot \frac{\partial J}{\partial \hat{y}}$$

$$\frac{\partial J}{\partial J} = 1$$

$$\frac{\partial J}{\partial z} = 2z \cdot \frac{\partial J}{\partial J}$$

$$\frac{\partial J}{\partial \hat{y}} = -1 \cdot \frac{\partial J}{\partial z}$$

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

$$\frac{\partial J}{\partial x} = w \cdot \frac{\partial J}{\partial q}$$

Backpropagation algorithm

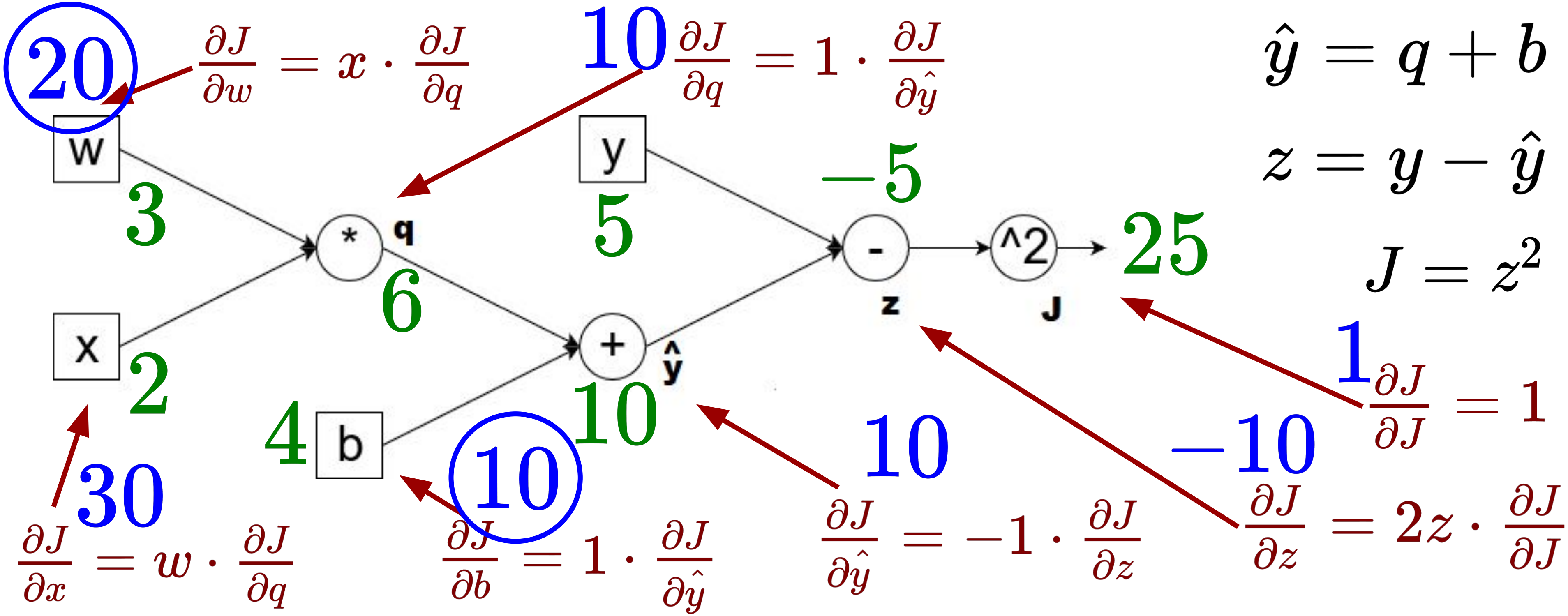
Step 5: Substitution - backpropagation

$$q = wx$$

$$\hat{y} = q + b$$

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

$$J = z^2$$



$$\frac{\partial J}{\partial w} = x \cdot \frac{\partial J}{\partial q}$$

$$\frac{\partial J}{\partial q} = 1 \cdot \frac{\partial J}{\partial \hat{y}}$$

$$\frac{\partial J}{\partial J} = 1$$

$$\frac{\partial J}{\partial z} = 2z \cdot \frac{\partial J}{\partial J}$$

$$\frac{\partial J}{\partial \hat{y}} = -1 \cdot \frac{\partial J}{\partial z}$$

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

$$\frac{\partial J}{\partial x} = w \cdot \frac{\partial J}{\partial q}$$

Backpropagation algorithm

Step 5: Substitution - backpropagation

$$\frac{\partial J}{\partial w} = 20$$

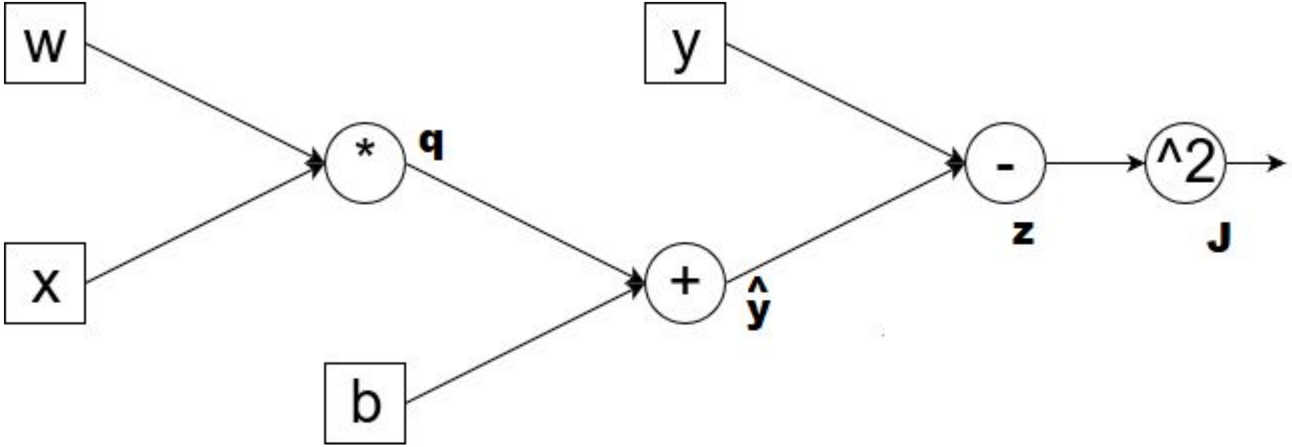
$$x = 2,\ w = 3,\ b = 4,\ y = 5$$

$$q = wx$$

$$\hat{y} = q + b$$

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

$$J = z^2$$



$$\frac{\partial J}{\partial b} = 10$$

Backpropagation algorithm

Step 6: Updating the weights / parameters

$$x = 2,\ w = 3,\ b = 4,\ y = 5$$

$$\frac{\partial J}{\partial w} = 20 \qquad \frac{\partial J}{\partial b} = 10$$

For example, let $\alpha = 0.1$

```
repeat until convergence   {
  for $\forall\, \theta \in \Theta$   {
    $grad_\theta = \frac{\partial}{\partial \theta} J(\Theta)$
  }
  for $\forall\, \theta \in \Theta$   {
    $\theta = \theta - \alpha\ grad_\theta$
  }
}
```

Backpropagation algorithm

Step 6: Updating the weights / parameters

$$x = 2,\ w = 3,\ b = 4,\ y = 5$$

$$\frac{\partial J}{\partial w} = \ 20 \qquad \frac{\partial J}{\partial b} = \ 10$$

$$w := 3 - 0.1 \cdot 20 = 1$$

$$b := 4 - 0.1 \cdot 10 = 3$$

```
repeat until convergence   {
  for $\forall\, \theta \in \Theta$   {
    $grad_\theta = \frac{\partial}{\partial\theta} J(\Theta)$
  }
  for $\forall\, \theta \in \Theta$   {
    $\theta = \theta - \alpha\, grad_\theta$
  }
}
```

$$\alpha = 0.1$$

Backpropagation algorithm

Step 6: Updating the weights / parameters

$$x = 2,\ w = 3,\ b = 4,\ y = 5$$

$$\frac{\partial J}{\partial w} = 20 \qquad \frac{\partial J}{\partial b} = 10$$

$$w := 3 - 0.1 \cdot 20 = 1$$

$$b := 4 - 0.1 \cdot 10 = 3$$

```
repeat until convergence   {
  for $\forall\,\theta \in \Theta$   {
    $grad_\theta = \frac{\partial}{\partial\theta} J(\Theta)$
  }
  for $\forall\,\theta \in \Theta$   {
    $\theta = \theta - \alpha\, grad_\theta$
  }
}
```

By computing the loss value with the new parameters, we can easily check whether the loss has decreased...

$$\alpha = 0.1$$

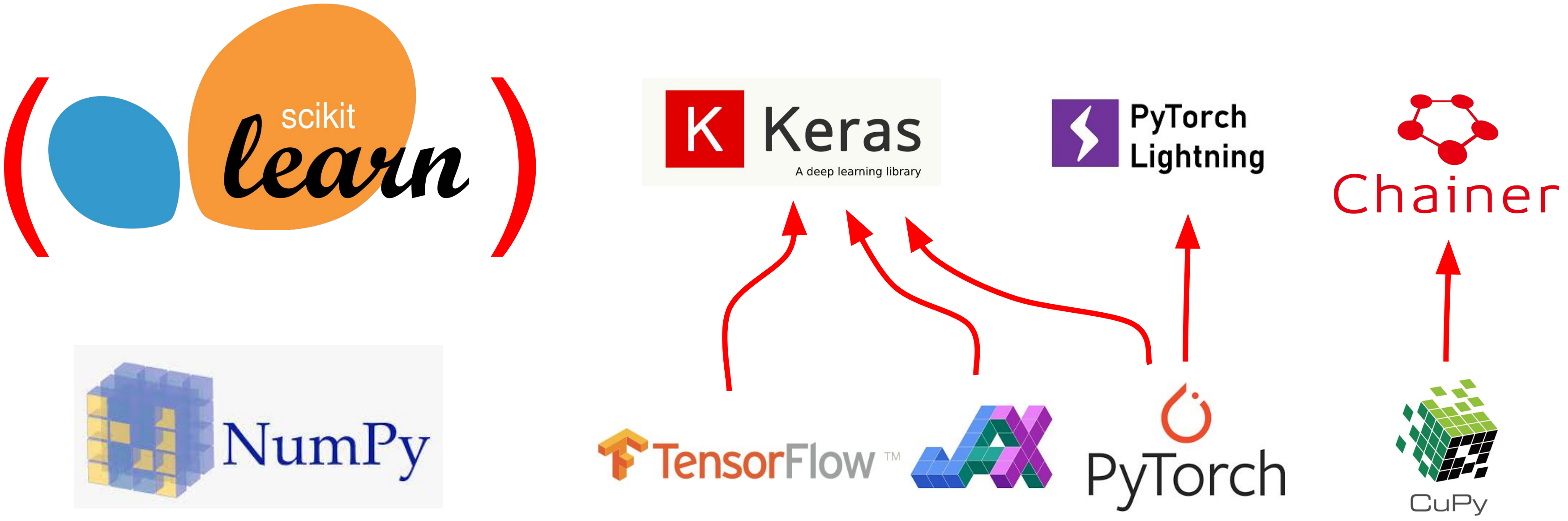
Backpropagation algorithm

Step 1: Symbolic derivation

Step 2: Parameter initialization

> **Step 3:** Choosing an example (or a mini-batch) from the training set
>
> **Step 4:** Substitution - feed forward
>
> **Step 5:** Substitution - backpropagation
>
> **Step 6:** Updating the parameters
>
> **GOTO STEP 3** - Stop after a given num. of iterations OR early stopping

Software for neural networks



CPU only

CPU & GPU
Computational graphs
Automatic differentiation

Reminder - Logistic regression in NumPy

```
intercept = np.ones((X.shape[0], 1))
X = np.concatenate((intercept, X), axis=1)
self.theta = np.zeros(X.shape[1])

for i in range(self.num_iter):
    h = self.__sigmoid(np.dot(X, self.theta))
    gradient = np.dot(X.T, (h - y)) / y.size
    self.theta -= self.lr * gradient
```

Reminder - Logistic regression in NumPy

```
intercept = np.ones((X.shape[0], 1))
X = np.concatenate((intercept, X), axis=1)
self.theta = np.zeros(X.shape[1])

for i in range(self.num_iter):
    h = self.__sigmoid(np.dot(X, self.theta))
    gradient = np.dot(X.T, (h - y)) / y.size
    self.theta -= self.lr * gradient
```

The **gradient formula** for each parameter (vector/matrix) **must be calculated manually**...

Besides, the **automatic reuse of** already calculated **gradients is not implemented** either.

Reminder - Logistic regression in NumPy

```
intercept = np.ones((X.shape[0], 1))
X = np.concatenate((intercept, X), axis=1)
self.theta = np.zeros(X.shape[1])

for i in range(self.num_iter):
    h = self.__sigmoid(np.dot(X, self.theta))
    gradient = np.dot(X.T, (h - y)) / y.size
    self.theta -= self.lr * gradient
```

Type of variables:

numpy.ndarray

Reminder - Logistic regression in NumPy

```
intercept = np.ones((X.shape[0], 1))
X = np.concatenate((intercept, X), axis=1)
self.theta = np.zeros(X.shape[1])

for i in range(self.num_iter):
    h = self.__sigmoid(np.dot(X, self.theta))
    gradient = np.dot(X.T, (h - y)) / y.size
    self.theta -= self.lr * gradient
```

Type of variables:
numpy.ndarray

NumPy ndarrays refer to specific memory locations.
A NumPy operation is evaluated immediately when called, and its result is stored in new/other ndarrays.
→ **Greedy evaluation**

TensorFlow (v1)

A classic library for neural networks: TensorFlow 1.x

TensorFlow (v1), MLP

```
w1 = tf.Variable(tf.random_normal([n_input, n_hidden]))
w2 = tf.Variable(tf.random_normal([n_hidden, n_output]))
b1 = tf.Variable(tf.random_normal([n_hidden]))
b2 = tf.Variable(tf.random_normal([n_output]))

X = tf.placeholder(tf.float32, [None, n_input])
Y = tf.placeholder(tf.float32, [None, n_output])

layer_1 = tf.sigmoid(tf.add(tf.matmul(X, w1), b1))
layer_2 = tf.add(tf.matmul(layer_1, w2), b2)

loss_op = tf.reduce_mean(tf.square(tf.subtract(layer_2, y)))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
train_op = optimizer.minimize(loss_op)
...
```

TensorFlow (v1), MLP

```
w1 = tf.Variable(tf.random_normal([n_input, n_hidden]))
w2 = tf.Variable(tf.random_normal([n_hidden, n_output]))
b1 = tf.Variable(tf.random_normal([n_hidden]))
b2 = tf.Variable(tf.random_normal([n_output]))

X = tf.placeholder(tf.float32, [None, n_input])
Y = tf.placeholder(tf.float32, [None, n_output])

layer_1 = tf.sigmoid(tf.add(tf.matmul(X, w1), b1))
layer_2 = tf.add(tf.matmul(layer_1, w2), b2)

loss_op = tf.reduce_mean(tf.square(tf.subtract(layer_2, y)))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
train_op = optimizer.minimize(loss_op)
...
```

Type: **tensorflow.Tensor**
(there are many more of this in this piece of code…)

TensorFlow (v1), MLP

Tf.Tensor

E.g.,: `h = tf.matmul(X, w1)` where **`h, X, w1`** are of Tensor types.

TensorFlow (v1), MLP

Tf.Tensor

E.g.,: `h = tf.matmul(X, w1)` where `h, X, w1` are of Tensor types.

The Tensor type is very similar to the NumPy ndarray type:
in some respects, it represents an array of values (n-dimensional).

On the other hand, however, `h` is not evaluated as a result of the above call. `h` therefore actually **represents the (potential) result of the** above matrix multiplication **operation.**

TensorFlow (v1), MLP

```
w1 = tf.Variable(tf.random_normal([n_input, n_hidden]))
w2 = tf.Variable(tf.random_normal([n_hidden, n_output]))
b1 = tf.Variable(tf.random_normal([n_hidden]))
b2 = tf.Variable(tf.random_normal([n_output]))

X = tf.placeholder(tf.float32, [None, n_input])
Y = tf.placeholder(tf.float32, [None, n_output])

layer_1 = tf.sigmoid(tf.add(tf.matmul(X, w1), b1))
layer_2 = tf.add(tf.matmul(layer_1, w2), b2)

loss_op = tf.reduce_mean(tf.square(tf.subtract(layer_2, y)))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
train_op = optimizer.minimize(loss_op)
...
```

Op(eration) **constructors**

TensorFlow (v1), MLP

Operation constructors

E.g.,:

```
h = tf.matmul(X, w1)
```

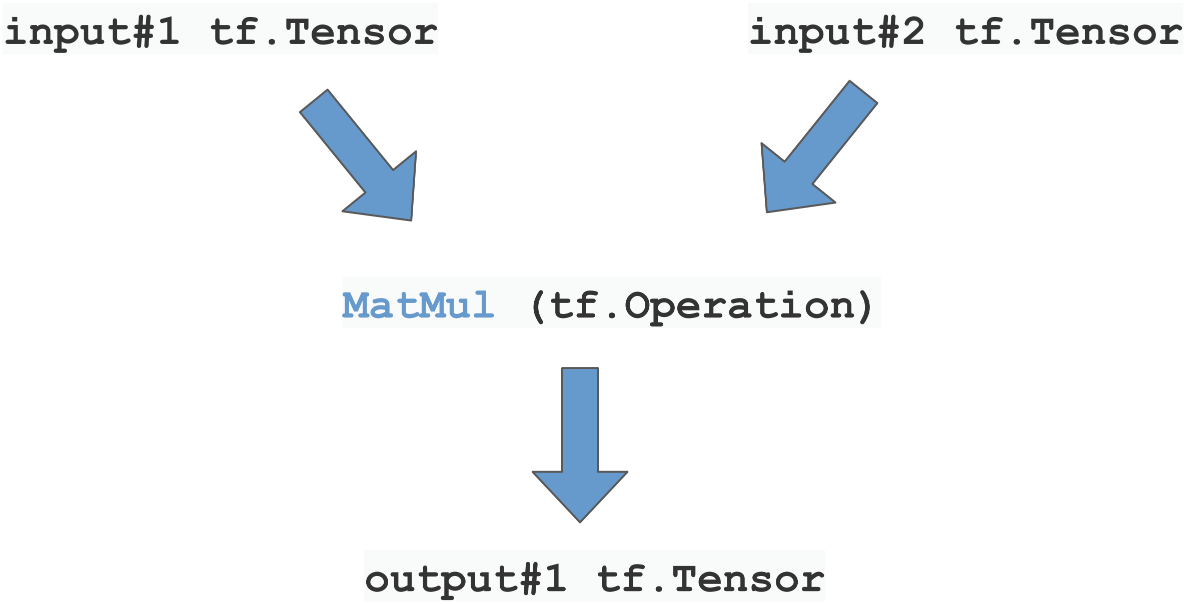
TensorFlow (v1), MLP

Operation constructors

E.g.,: `h = tf.matmul(X, w1)`

The above call creates a **tf.Operation** type object (here MatMul type), which represents an operation (here matrix multiplication).
It attaches the Tensor objects referred by `x` and `w1` as inputs to the operation and returns the Tensor object attached to the operation, given as the value of `h`, which represents the output of the operation.

TensorFlow (v1) - Computational graph



input#1 tf.Tensor

input#2 tf.Tensor

MatMul (tf.Operation)

output#1 tf.Tensor

```
h = tf.matmul(X, w1)
```

TensorFlow (v1) - Computational graph

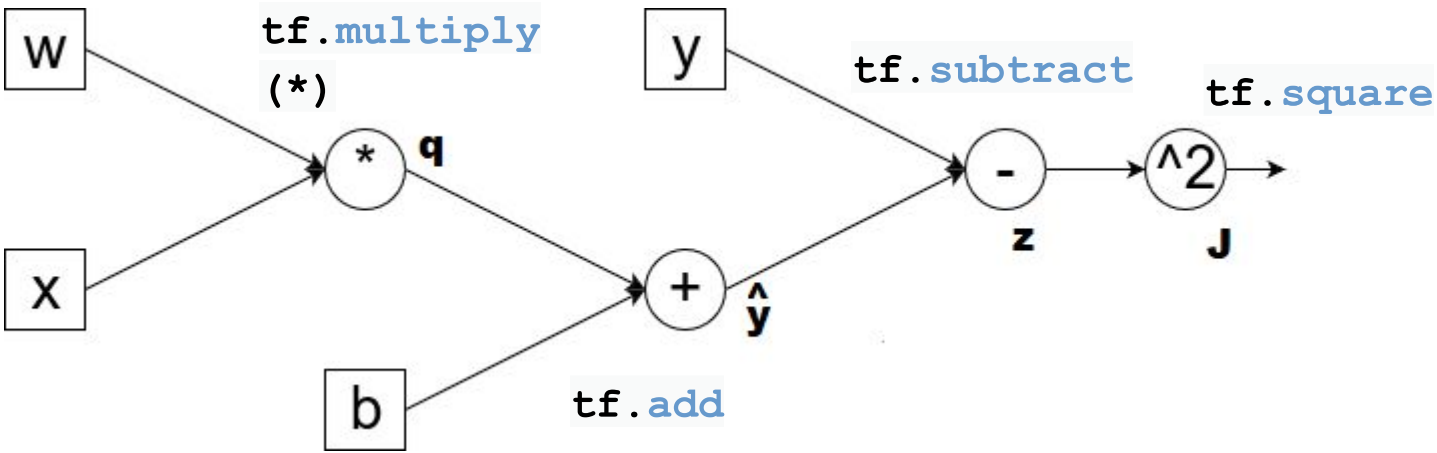
```
...
layer_1 = tf.sigmoid(tf.add(tf.matmul(X, w1), b1))
layer_2 = tf.add(tf.matmul(layer_1, w2), b2)

loss_op = tf.reduce_mean(tf.square(tf.subtract(layer_2, y)))
...
```

The operations, linked together, form a graph.
The vertices of the graph **are the operations** (tf.Operation)**, and the edges are the tensors** (tf.Tensor) **that store the intermediate, temporary results.**

TensorFlow (v1) - Computational graph

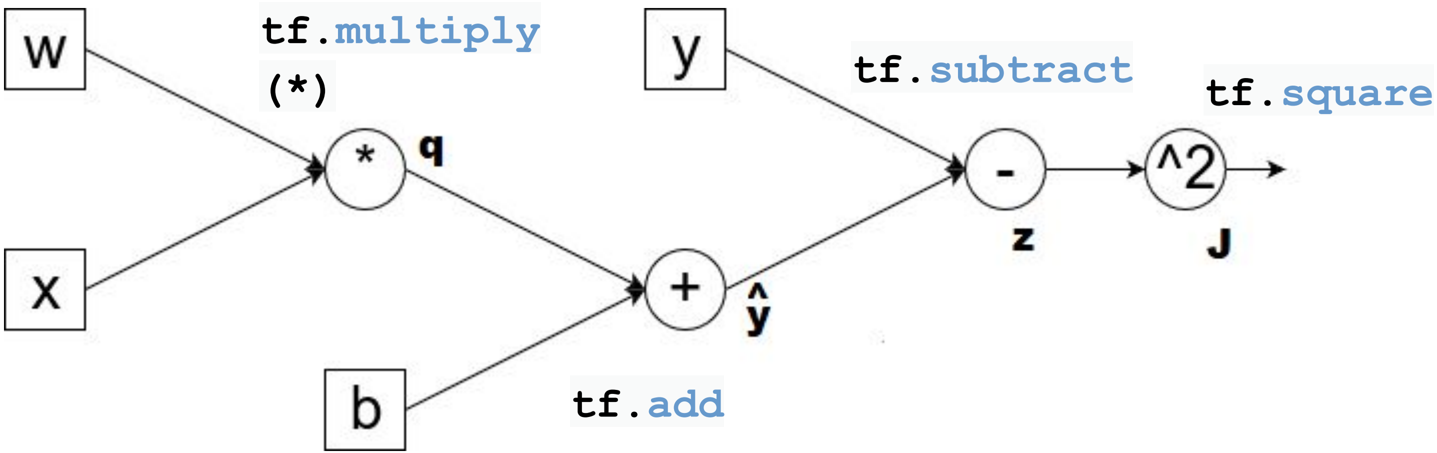
Computational graph with the corresponding TensorFlow Op constructors



(Not the computational graph associated with the previous code)

TensorFlow (v1) - Computational graph

Computational graph with the corresponding TensorFlow Op constructors



(*) tf.multiply is for scalars only, for matrices we can use tf.matmul.

TensorFlow (v1), MLP

```
w1 = tf.Variable(tf.random_normal([n_input, n_hidden]))
w2 = tf.Variable(tf.random_normal([n_hidden, n_output]))
b1 = tf.Variable(tf.random_normal([n_hidden]))
b2 = tf.Variable(tf.random_normal([n_output]))

X = tf.placeholder(tf.float32, [None, n_input])
Y = tf.placeholder(tf.float32, [None, n_output])

layer_1 = tf.sigmoid(tf.add(tf.matmul(X, w1), b1))
layer_2 = tf.add(tf.matmul(layer_1, w2), b2)

loss_op = tf.reduce_mean(tf.square(tf.subtract(layer_2, y)))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
train_op = optimizer.minimize(loss_op)
...
```

Type: **tensorflow.Variable**

TensorFlow (v1), MLP

tf.Variable

E.g.,:

```
b2 = tf.Variable(tf.random_normal([n_output]))
```

TensorFlow (v1), MLP

tf.Variable

E.g.,:

```
b2 = tf.Variable(tf.random_normal([n_output]))
```

tf.Variable objects are similar to Tensor objects, but their state is **persistent** and their values can be modified by various operations.
For this reason, we declare the parameters of neural networks with the **tf.Variable** type and modify their values during gradient backpropagation.

TensorFlow (v1), MLP

```
w1 = tf.Variable(tf.random_normal([n_input, n_hidden]))
w2 = tf.Variable(tf.random_normal([n_hidden, n_output]))
b1 = tf.Variable(tf.random_normal([n_hidden]))
b2 = tf.Variable(tf.random_normal([n_output]))

X = tf.placeholder(tf.float32, [None, n_input])
Y = tf.placeholder(tf.float32, [None, n_output])

layer_1 = tf.sigmoid(tf.add(tf.matmul(X, w1), b1))
layer_2 = tf.add(tf.matmul(layer_1, w2), b2)

loss_op = tf.reduce_mean(tf.square(tf.subtract(layer_2, y)))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
train_op = optimizer.minimize(loss_op)
...
```

Placeholder Tensor objects

(Type: tensorflow.Tensor)

TensorFlow (v1), MLP

Placeholder Tensor constructor (only in TF v1)

E.g.,:

```
tf.placeholder(tf.float32, [None, n_input])
```

TensorFlow (v1), MLP

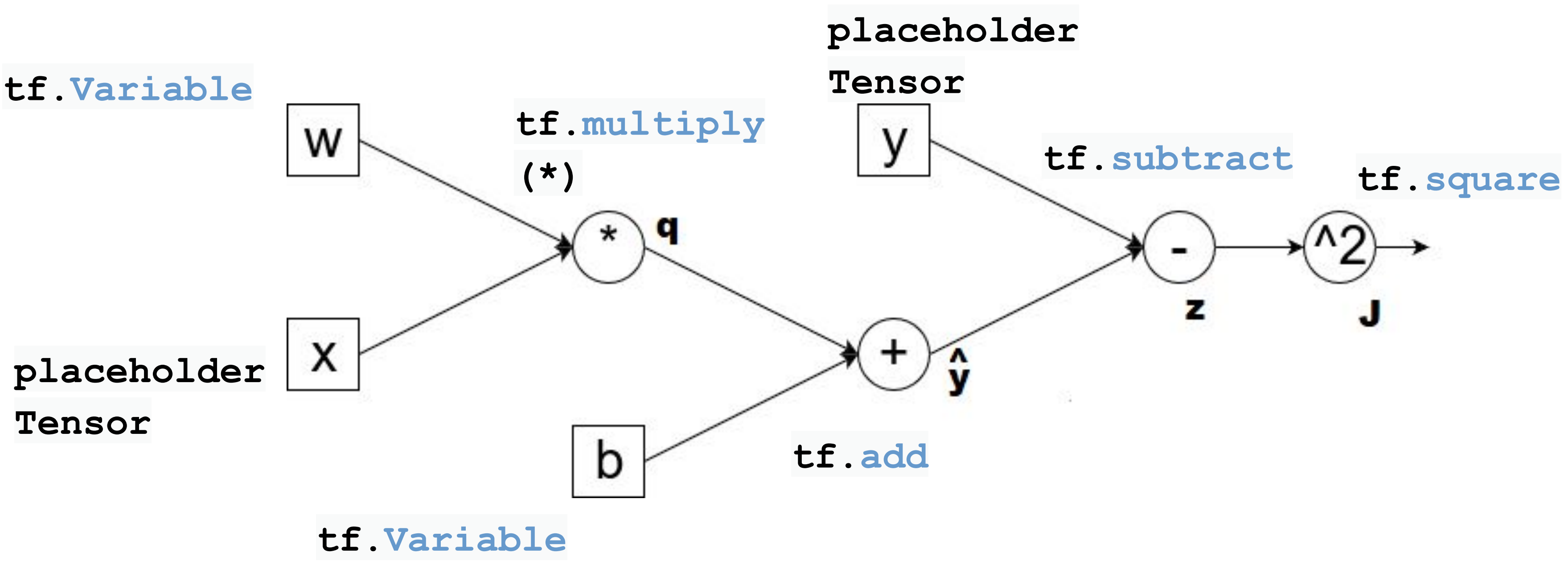
Placeholder Tensor constructor (only in TF v1)

E.g.,:

```
tf.placeholder(tf.float32, [None, n_input])
```

Placeholders are Tensor objects whose **values** are not obtained from the output of operations, but **are specified explicitly.**
The input/output variables of the neural network will therefore be as follows.

TensorFlow (v1) - Computational graph



tf.Variable

placeholder
Tensor

tf.multiply
(*)

placeholder
Tensor

tf.subtract

tf.square

tf.add

tf.Variable

TensorFlow (v1), MLP

```
...
with tf.Session() as sess:
    # ...
    for i in range(total_batch):
        batch_x, batch_y = next(my_batch_iterator)
        _, loss_npy = sess.run([train_op, loss_op], feed_dict={X: batch_x,
                                                                Y: batch_y})
```

TensorFlow (v1), MLP

```
...
with tf.Session() as sess:
    # ...
    for i in range(total_batch):
        batch_x, batch_y = next(my_batch_iterator)
        _, loss_npy = sess.run([train_op, loss_op], feed_dict={X: batch_x,
                                                                Y: batch_y})
```

The **previously defined loss_op** and **train_op** graphs have only existed in symbolic form so far, and **will be evaluated here for the first time:** We feed the next batch of training samples into the placeholder tensors.

tf.Session (only in TF v1)**:** A runtime environment attached to physical hardware

The specific value of the cost will be calculated during evaluation.
The result will be returned in a NumPy array.

TensorFlow (v1) - Computational graph

Why is it necessary to construct a computational graph,
and what are the **advantages of asynchronous / lazy evaluation?**

TensorFlow (v1) - Computational graph

Why is it necessary to construct a computational graph,
and what are the **advantages of asynchronous / lazy evaluation?**

- **Evaluation may be accelerated** by rearranging the order of independent operations (**out-of-order execution**).

TensorFlow (v1) - Computational graph

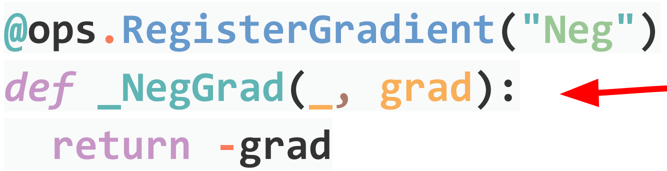
Why is it necessary to construct a computational graph,
and what are the **advantages of asynchronous / lazy evaluation?**

- **Evaluation may be accelerated** by rearranging the order of independent operations (**out-of-order execution**).
- **Automatic differentation.**

TensorFlow (v1) - Automatic differentiation

For every elementary operation that we want to use in an expression optimized by gradient descent (e.g., neural network), **the derivative of this operation must be specified.**

Usually, we do not need to define new elementary operations, as many of them are already implemented in TensorFlow.



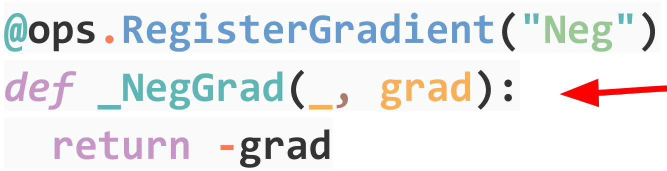
```
@ops.RegisterGradient("Neg")
def _NegGrad(_, grad):
  return -grad
```

The gradient of the negation op
in file: tensorflow/python/ops/
math_grad.py

TensorFlow (v1) - Automatic differentiation

For every elementary operation that we want to use in an expression optimized by gradient descent (e.g., neural network), **the derivative of this operation must be specified.**

Usually, we do not need to define new elementary operations, as many of them are already implemented in TensorFlow.



```
@ops.RegisterGradient("Neg")
def _NegGrad(_, grad):
  return -grad
```

We negate the gradient coming from the output and pass it towards the input (the derivative of negation is negation).

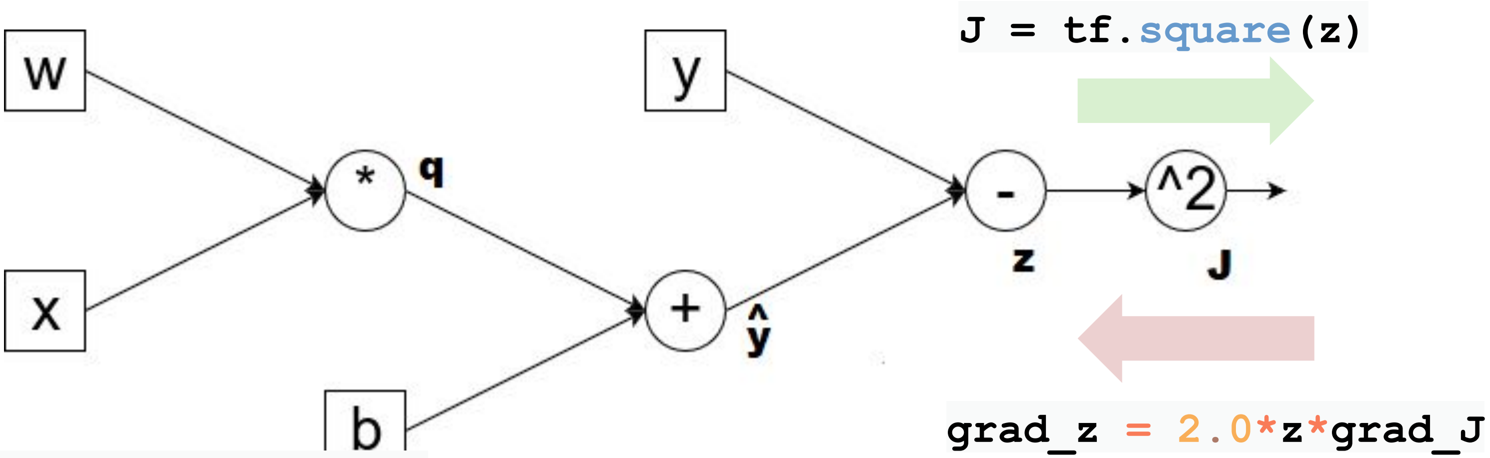
The gradient of the negation op in file: tensorflow/python/ops/math_grad.py

TensorFlow (v1) - Automatic differentiation

Gradient of the “Square” operation:

```
@ops.RegisterGradient("Square")
def _SquareGrad(op, grad):
  x = op.inputs[0]
  ...
  return grad * (2.0 * x)
```

TensorFlow (v1) - Computational graph



```
J = tf.square(z)
```

```
grad_z = 2.0*z*grad_J
```

```
@ops.RegisterGradient("Square")
def _SquareGrad(op, grad):
  x = op.inputs[0]
  ...
  return grad * (2.0 * x)
```

TensorFlow (v1) - Computational graph

What are the **disadvantages of asynchronous / lazy evaluation?**

TensorFlow (v1) - Computational graph

What are the **disadvantages of asynchronous / lazy evaluation?**

- The **Python interpreter cannot be used for debugging.** Debugging with the use of Print nodes and callbacks connected to the graph.
- The **compilation** procedure **takes time.**
- **Constructing** / modifying **the computation graph during runtime is complicated.**

PyTorch

- By default (mostly) **eager evaluation** (Eager mode),
  lazy evaluation on demand (Graph mode, torch.jit).
- Computational **graph construction is only necessary if we want to propagate gradients** through the operations in question.
- **Dynamic graph construction.**

PyTorch

```
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(my_model.parameters(), lr=learning_rate)

for x, y in my_dataloader:
  optimizer.zero_grad()
  y_pred = my_model(x)
  loss = loss_fn(y_pred, y)
  loss.backward()
  optimizer.step()
```

PyTorch

```
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(my_model.parameters(), lr=learning_rate)

for x, y in my_dataloader:
  optimizer.zero_grad()
  y_pred = my_model(x)
  loss = loss_fn(y_pred, y)
  loss.backward()
  optimizer.step()
```

The **computational graph** is also present here. We want to minimize the `loss` expression, so gradient backpropagation will start from there in the graph, as in TensorFlow, passing through the `my_model` hypothesis function.

The `torch.Tensor.backward()` call performs **gradient backpropagation** and calculates the partial derivatives of the loss expression according to the intermediate tensors of the computational graph.

PyTorch

```
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(my_model.parameters(), lr=learning_rate)

losses = []
for x, y in my_dataloader:
  optimizer.zero_grad()
  y_pred = my_model(x)
  loss = loss_fn(y_pred, y)
  loss.backward()
  optimizer.step()
  losses.append(loss.detach())
```

PyTorch

```
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(my_model.parameters(), lr=learning_rate)

losses = []
for x, y in my_dataloader:
  optimizer.zero_grad()
  y_pred = my_model(x)
  loss = loss_fn(y_pred, y)
  loss.backward()
  optimizer.step()
  losses.append(loss.detach())
```

PyTorch uses greedy evaluation by default, so (unlike TensorFlow 1.x) **we only need to use computational graphs where we want to propagate gradients.**

We want to record the value of the loss function in each iteration, for example, for later plotting of the loss curves. For this, it is not necessary to maintain the computational graphs from each iteration; the loss value itself is sufficient.

The `torch.Tensor.detach()` function returns a view of the `loss` tensor (scalar), which can be used in further operations **without the continued building of the computational graph.** As the graph is not built, gradients will not pass through the `detach()` operation.

PyTorch

```
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(my_model.parameters(), lr=learning_rate)

losses = []
for x, y in my_dataloader:
  optimizer.zero_grad()
  y_pred = my_model(x)
  loss = loss_fn(y_pred, y)
  loss.backward()
  optimizer.step()
  losses.append(loss.detach().numpy())
```

PyTorch

```
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(my_model.parameters(), lr=learning_rate)

losses = []
for x, y in my_dataloader:
  optimizer.zero_grad()
  y_pred = my_model(x)
  loss = loss_fn(y_pred, y)
  loss.backward()
  optimizer.step()
  losses.append(loss.detach().numpy())
```

A tensor that is not bound to a computational graph (and `device=′cpu′`) **can** also **be converted to a NumPy array** (ndarray) type. In this case, the tensor and the NumPy array share memory.

PyTorch

```
@torch.no_grad()
def accuracy_binary_torch(ys_pred, ys_true):
  return torch.mean((torch.round(ys_pred) == ys_true).to(torch.float32))

for x, y in my_dataloader:
  optimizer.zero_grad()
  y_pred = my_model(x)
  loss = loss_fn(y_pred, y)
  loss.backward()
  optimizer.step()
  acc = accuracy_binary_torch(y_pred, y)
```

PyTorch

Alternative to `torch.Tensor.detach()`: **Any new tensors created in the body of a function decorated with `@torch.no_grad()` will be independent of the computational graph** and cannot be used for backpropgation.

```
@torch.no_grad()
def accuracy_binary_torch(ys_pred, ys_true):
  return torch.mean((torch.round(ys_pred) == ys_true).to(torch.float32))

for x, y in my_dataloader:
  optimizer.zero_grad()
  y_pred = my_model(x)
  loss = loss_fn(y_pred, y)
  loss.backward()
  optimizer.step()
  acc = accuracy_binary_torch(y_pred, y)
```

Since we do not optimize according to the accuracy metric, but only use it for logging the model performance, **we save time and memory by disabling graph building and gradient computation in this function.**

PyTorch

```
def accuracy_binary_torch(ys_pred, ys_true):
  return torch.mean((torch.round(ys_pred) == ys_true).to(torch.float32))

for x, y in my_dataloader:
  optimizer.zero_grad()
  y_pred = my_model(x)
  loss = loss_fn(y_pred, y)
  loss.backward()
  optimizer.step()
  with torch.no_grad():
    acc = accuracy_binary_torch(y_pred, y)
```

PyTorch

```python
def accuracy_binary_torch(ys_pred, ys_true):
  return torch.mean((torch.round(ys_pred) == ys_true).to(torch.float32))

for x, y in my_dataloader:
  optimizer.zero_grad()
  y_pred = my_model(x)
  loss = loss_fn(y_pred, y)
  loss.backward()
  optimizer.step()
  with torch.no_grad():
    acc = accuracy_binary_torch(y_pred, y)
```

The `torch.no_grad()` block works in a similar way...