AdaLinE

Neuroinformatics Tutorial 7

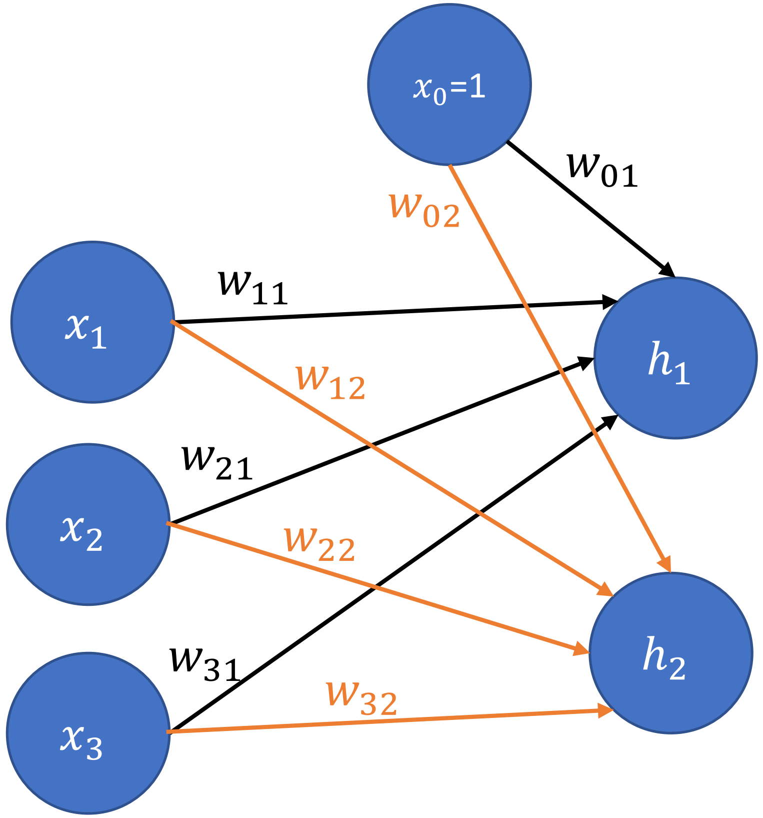
Duc Duy Pham[1]

[1]Intelligent Systems, Faculty of Engineering,
University of Duisburg-Essen, Germany

Content

- **Revision: Practical Task**
- Revision: Lecture
- Tensorflow
- New Practical Task

Calculation of propagated value



$$h_1 = \sum_{i=0}^{3} w_{i1} x_i$$

$$h_2 = \sum_{i=0}^{3} w_{i2} x_i$$

$$x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \qquad W = \begin{bmatrix} w_{01} & w_{02} \\ w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix} \qquad \begin{bmatrix} h_1 \\ h_2 \end{bmatrix} = W^T \cdot x$$

Content

- Revision: Practical Task
- **Revision: Lecture**
- Tensorflow
- New Practical Task

Revision: Lecture

- What are the main components of AdaLinE? (must know!)

Revision: Lecture

- What are the main components of AdaLinE? (must know!)
  - Real input: $(1, x_1, \ldots, x_n)^T \in \mathbb{R}^{n+1}$
  - Real weights $(-\Theta, w_1, \ldots, w_n)^T \in \mathbb{R}^n$
  - Propagation function (linear associator) $\sum_{i=0}^{n} w_i x_i$
  - Activation function: Identity!

Revision: Lecture

- Which statements regarding AdaLinE and RBP are true?
    1. AdaLinE is capable of regression
    2. RBP has same activation function as AdaLinE
    3. AdaLinE has same structure as RBP except for propagation function
    4. RBP has same structure as AdaLinE except for activation function

Proportional Learning Rule

- How does the Proportional Learning Rule/Algorithm work? (must know!)

Proportional Learning Rule

- How does the Proportional Learning Rule/Algorithm work? (must know!)
  - Let $w := (-\Theta, w_1, \ldots, w_n)^T \in \mathbb{R}^{n+1}$ denote the extended weight vector including the bias
  - Let $w(i)$ denote the weight vector at iteration i
  - Let $x := (1, x_1, \ldots, x_n)^T \in \Omega \subset \mathbb{R}^{n+1}$ denote an arbitrary extended sample point from the training data set

Proportional Learning Rule

- Let $\hat{y}(x) \in \mathbb{R}$ denote the desired target output
- Let $\tilde{y}_{w(i)}(x) := f_a(f_p(x))$ denote the actual output of the AdaLinE with weight vector $w(i)$

Proportional Learning Rule

- Idea:
  - Draw a sample point $\boldsymbol{x}$ randomly
  - Check if AdaLinE output is target output
  - If not, adjust the weights!
    - Calculate error (difference in desired output)

$$\rho(x,i) := \hat{y}(x) - \tilde{y}_{w(i)}(x)$$

    - Add fraction (depending on learning rate and error) of sample $\boldsymbol{x}$ to current weight vector!

Proportional Learning Rule

- If $\hat{y}(x) == \tilde{y}_{w(i)}(x)$

  Do nothing

- If $\hat{y}(x) \neq \tilde{y}_{w(i)}(x)$

$$w(i+1) \leftarrow w(i) + \alpha \frac{\rho(x,i)x}{||x||^2}$$

Proportional Learning Rule

- We can generalize the weight update rule to:

$$w(i+1) \leftarrow w(i) + \Delta w(i)$$

- Therefore the amount of error reduction for AdaLinE is:

$$\begin{aligned}
|\Delta\rho(x,i)| &:= |\rho(x,i+1) - \rho(x,i)| \\
&= |[\hat{y}(x) - \tilde{y}_{w(i+1)}(x)] - [\hat{y}(x) - \tilde{y}_{w(i)}(x)]| \\
&= |[\hat{y}(x) - w(i+1)^T x] - [\hat{y}(x) - w(i)^T x]| \\
&= |-(w(i+1) - w(i))^T x| \\
&= |\Delta w(i)^T x|
\end{aligned}$$

Amount of error correction dependent on weight update!

Proportional Learning Rule

- There are many possible ways to choose the weight update, such that amount of error correction is the same

$$|\Delta\rho(x,i)| = |\Delta w(i)^T x|$$

- For proportional learning rule we choose weight update $\Delta w(i)$ such that $\Delta w(i)$ is parallel to sample point x , i.e.

$$\Delta w(i) := \gamma x, \quad \gamma \in \mathbb{R}$$

$$\left(\Delta w(i) := \alpha \frac{\rho(x,i)x}{||x||^2}, \quad \alpha \in \mathbb{R} \right)$$

- Why?

Previous Learning Achievements

- Learning achievement is encoded in $w(i)$
- We want $||\Delta w(i)||$ to be small

- Claim:
  - If $\Delta w(i)$ is parallel to x , i.e. $\Delta w(i) := \gamma x, \quad \gamma \in \mathbb{R}$

    then $||\gamma x|| \leq ||\Delta \tilde{w}||$

    for any weight update $\Delta \tilde{w}$
    with the same error reduction

$$|\Delta \tilde{w}^T x| = |\Delta \rho(x, i)| =: \zeta$$

Previous Learning Achievements

- Learning achievement is encoded in $w(i)$
- We want $||\Delta w(i)||$ to be small

- Claim:
  - If $\Delta w(i)$ is parallel to x, i.e. $\Delta w(i) := \gamma x, \quad \gamma \in \mathbb{R}$

    then $||\gamma x|| \leq ||\Delta \tilde{w}||$

    for any weight update $\Delta \tilde{w}$
    with the same error reduction

$$|\Delta \tilde{w}^T x| = |\Delta \rho(x, i)| =: \zeta$$

To prove this claim we will use the Cauchy Schwarz Inequality:

$$|\Delta \tilde{w}^T x|^2 \leq ||\Delta \tilde{w}||^2 ||x||^2$$

Previous Learning Achievements

- Directly from Cauchy Schwarz

$$\zeta^2 = |\Delta\tilde{w}^T x|^2 \leq ||\Delta\tilde{w}||^2 ||x||^2$$

- By design (choose scalar factor accordingly):

$$\zeta^2 = |\gamma x^T x|^2 = ||\gamma x||^2 ||x||^2$$

- Therefore:

$$||\gamma x||^2 ||x||^2 \leq ||\Delta\tilde{w}||^2 ||x||^2$$

- Finally:

$$||\gamma x|| \leq ||\Delta\tilde{w}||$$

Reminder

- For AdaLinE the weight update is defined as:

$$\Delta w(i) := \alpha \frac{\rho(x,i)x}{||x||^2}, \quad \alpha \in \mathbb{R}$$

- I.e. parallel to x !

Revision: Lecture

- Alternative to Proportional Learning rule?
  - Gradient Descent on some loss function
    (In Lecture: MSE)

  - Basically any optimization approach could work!
    - Optimization Problem:
      Find weight vector, such that loss is minimal

Drawing

Content

- Revision: Practical Task
- Revision: Lecture
- Tensorflow
- New Practical Task

Tensorflow

- Deep Learning Library (from Google)
- Widely used for Deep Learning Applications
- Some (popular) alternatives:
  - PyTorch (Facebook, before: (also) NYU)
  - Caffe2 (Facebook, before: UC Berkeley)
  - CNTK (Microsoft)
  - MXNet (U Washington, MIT, Hong Kong U, etc..., associated with Amazon)
  - Theano (U Montréal -> development discontinued)
  - Keras (High Level Interface for Tensorflow, CNTK, Theano, MXNet)
    - Integrated in Tensorflow >= 2.0

Tensorflow

- Works with computational graphs
- Processes tensors

Computational Graph

- ```
  x = 5;
  y = 4;
  w = 3;
  a = x-y;
  b = y*w;
  c = b+w;
  d = a*c;
  ```
  ```

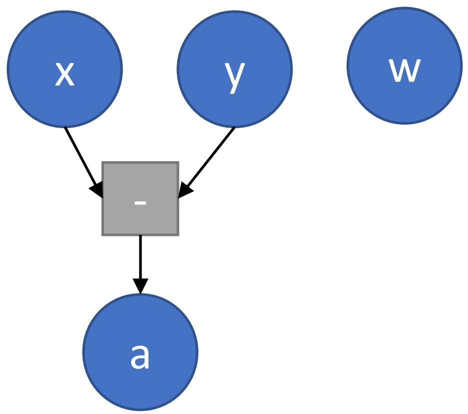
Computational Graph

- ```
  x = 5;
  y = 4;
  w = 3;
  a = x-y;
  b = y*w;
  c = b+w;
  d = a*c;
  ```


  ```

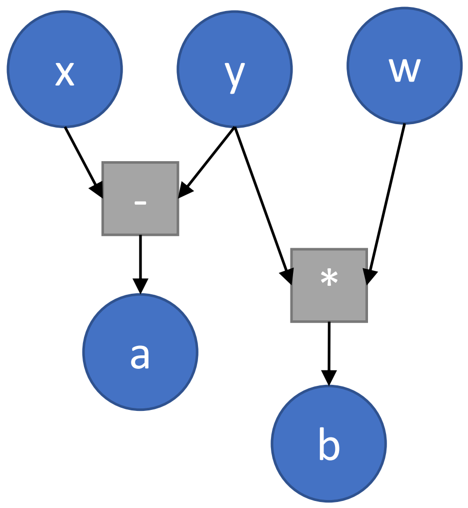
Computational Graph

- ```
  x = 5;
  y = 4;
  w = 3;
  a = x-y;
  b = y*w;
  c = b+w;
  d = a*c;
  ```


  ```

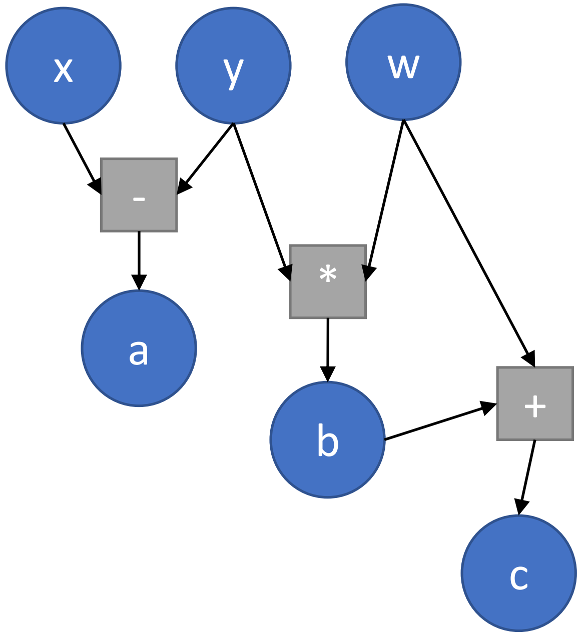
Computational Graph

- ```
  x = 5;
  y = 4;
  w = 3;
  a = x-y;
  b = y*w;
  c = b+w;
  d = a*c;
  ```


  ```

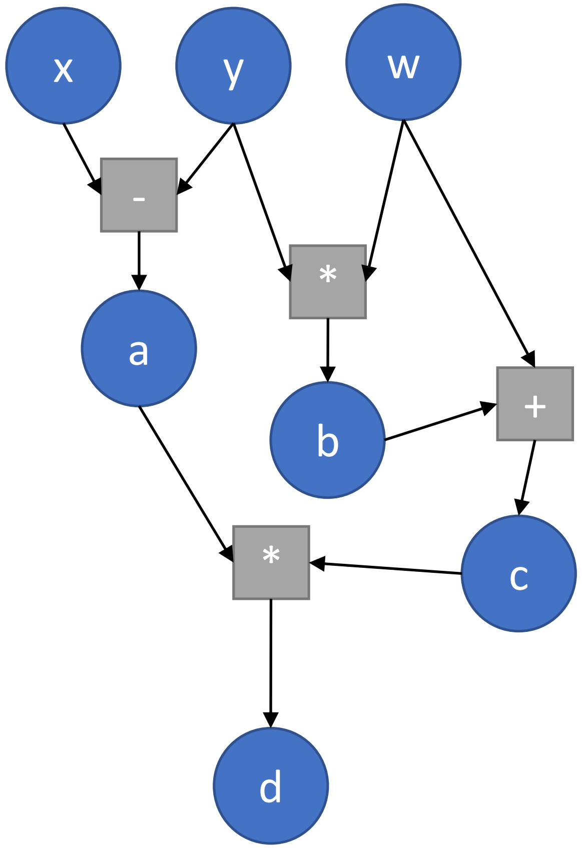
Computational Graph

- ```
  x = 5;
  y = 4;
  w = 3;
  a = x-y;
  b = y*w;
  c = b+w;
  d = a*c;
  ```


  ```

Computational Graph

```
x = 5;
y = 4;
w = 3;
a = x-y;
b = y*w;
c = b+w;
d = a*c;
```



Computational Graph – tf.Variable()

- ```
  x    = np.array([[1.0],[2.1],[3.1]])
  W_t0 = tf.Variable(np.random.rand(3,3))
  h    = tf.matmul(W_t0, x)
  ```
  ```

Computational Graph – tf.Variable()

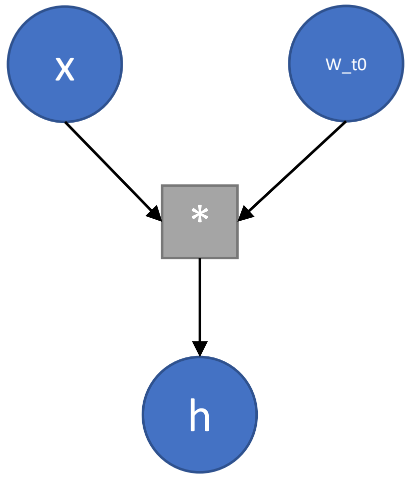
```
x    = np.array([[1.0],[2.1],[3.1]])
W_t0 = tf.Variable(np.random.rand(3,3))
h    = tf.matmul(W_t0, x)
```

Computational Graph – tf.Variable()

```
x    = np.array([[1.0],[2.1],[3.1]])
W_t0 = tf.Variable(np.random.rand(3,3))
h    = tf.matmul(W_t0, x)
```

Computational Graph – tf.Variable()

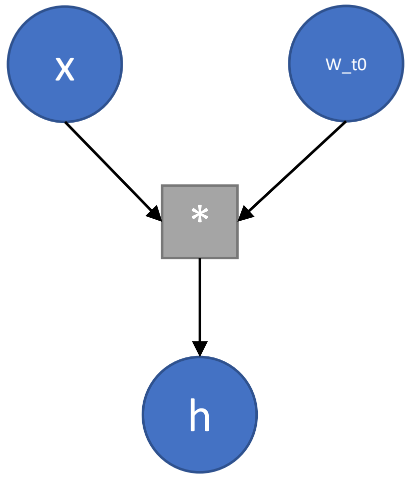
- ```
  x    = np.array([[1.0],[2.1],[3.1]])
  W_t0 = tf.Variable(np.random.rand(3,3))
  h    = tf.matmul(W_t0, x)
  ```


  ```

Computational Graph – tf.Variable()

- ```
  x    = np.array([[1.0],[2.1],[3.1]])      #3x1
  W_t0 = tf.Variable(np.random.rand(3,3))   #3x3
  h    = tf.matmul(W_t0, x)                 #3x1

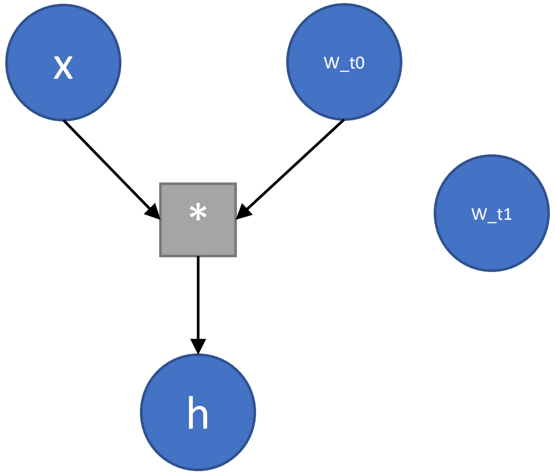
  W_t1 = tf.Variable(np.random.rand(1,3))   #1x3
  y    = tf.matmul(W_t1, x)                 #1x1
  ```


  ```

Computational Graph – tf.Variable()

- ```
  x    = np.array([[1.0],[2.1],[3.1]])      #3x1
  W_t0 = tf.Variable(np.random.rand(3,3))   #3x3
  h    = tf.matmul(W_t0, x)                 #3x1

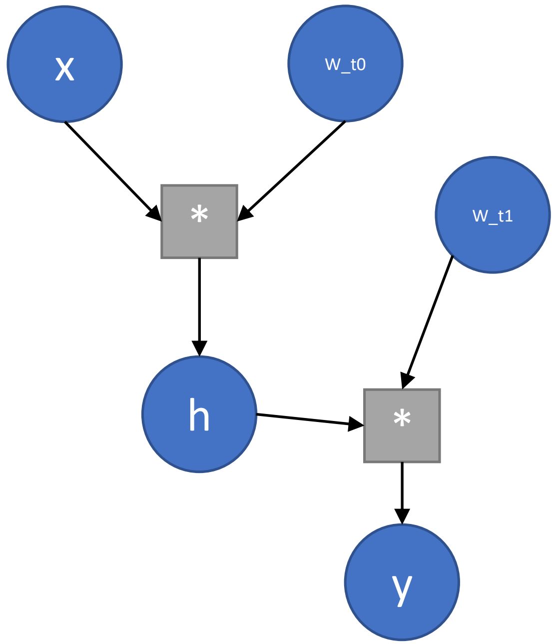
  W_t1 = tf.Variable(np.random.rand(1,3))   #1x3
  y    = tf.matmul(W_t1, x)                 #1x1
  ```


  ```

Computational Graph – tf.Variable()

```
x    = np.array([[1.0],[2.1],[3.1]])       #3x1
W_t0 = tf.Variable(np.random.rand(3,3))    #3x3
h    = tf.matmul(W_t0, x)                  #3x1

W_t1 = tf.Variable(np.random.rand(1,3))    #1x3
y    = tf.matmul(W_t1, x)                  #1x1
```



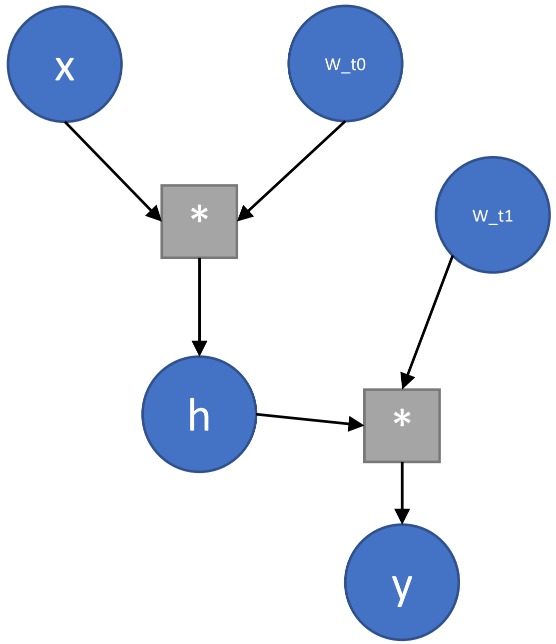
Computational Graph – loss

```
x    = np.array([[1.0],[2.1],[3.1]])      #3x1
W_t0 = tf.Variable(np.random.rand(3,3))   #3x3
h    = tf.matmul(W_t0, x)                 #3x1

W_t1 = tf.Variable(np.random.rand(1,3))   #1x3
y    = tf.matmul(W_t1, x)                 #1x1



y_   = np.array([[3.3]])
loss = tf.abs(y-y_)
```

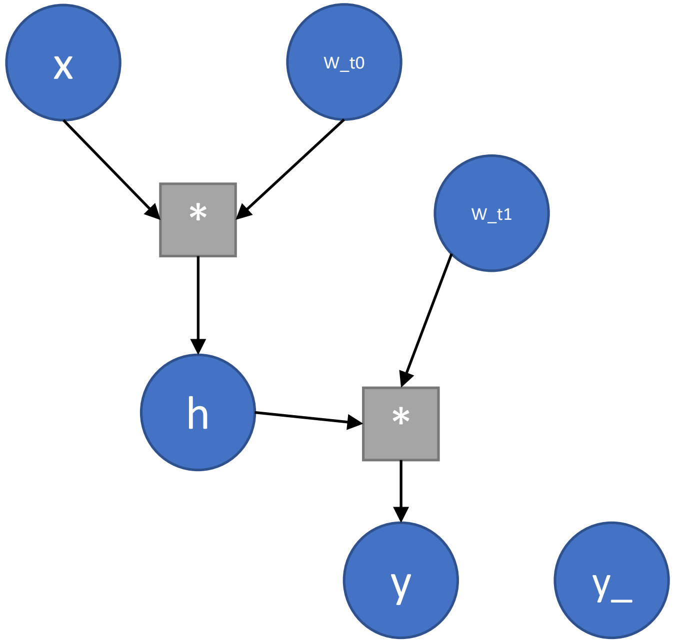


Computational Graph – loss

```
x    = np.array([[1.0],[2.1],[3.1]])      #3x1
W_t0 = tf.Variable(np.random.rand(3,3))   #3x3
h    = tf.matmul(W_t0, x)                 #3x1

W_t1 = tf.Variable(np.random.rand(1,3))   #1x3
y    = tf.matmul(W_t1, x)                 #1x1


y_   = np.array([[3.3]])
loss = tf.abs(y-y_)
```



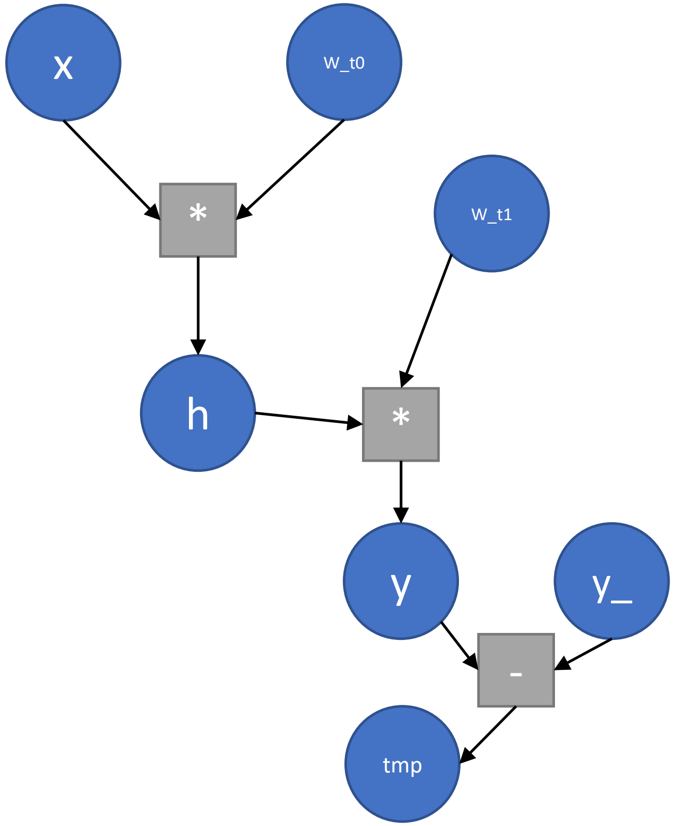
Computational Graph – loss

- ```
  x    = np.array([[1.0],[2.1],[3.1]])      #3x1
  W_t0 = tf.Variable(np.random.rand(3,3))   #3x3
  h    = tf.matmul(W_t0, x)                 #3x1

  W_t1 = tf.Variable(np.random.rand(1,3))   #1x3
  y    = tf.matmul(W_t1, x)                 #1x1



  y_   = np.array([[3.3]])
  loss = tf.abs(y-y_)
  ```


  ```

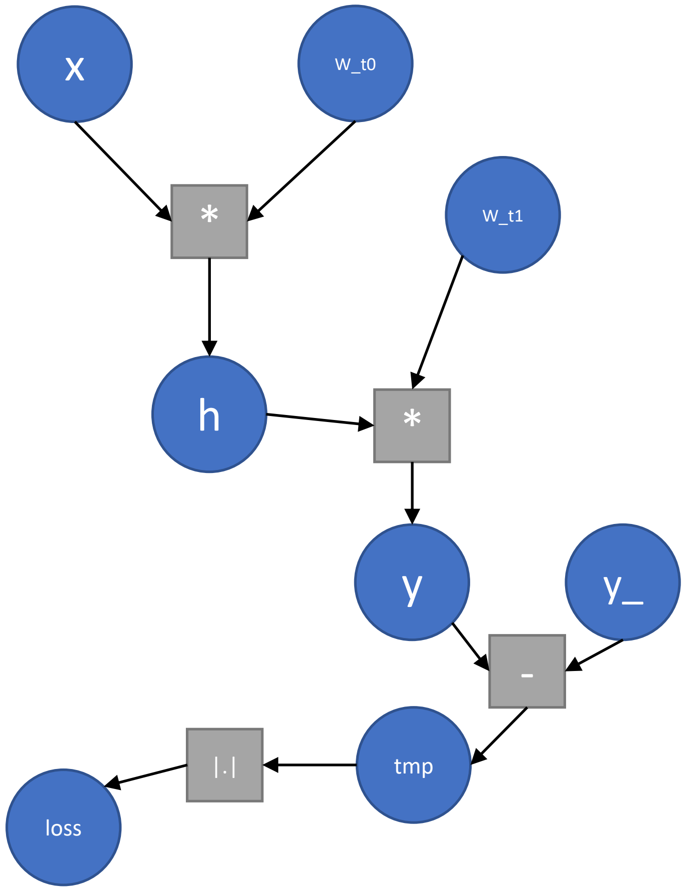
Computational Graph – loss

```
x    = np.array([[1.0],[2.1],[3.1]])      #3x1
W_t0 = tf.Variable(np.random.rand(3,3))   #3x3
h    = tf.matmul(W_t0, x)                 #3x1

W_t1 = tf.Variable(np.random.rand(1,3))   #1x3
y    = tf.matmul(W_t1, x)                 #1x1



y_   = np.array([[3.3]])
loss = tf.abs(y-y_)
```



Computational Graph – loss

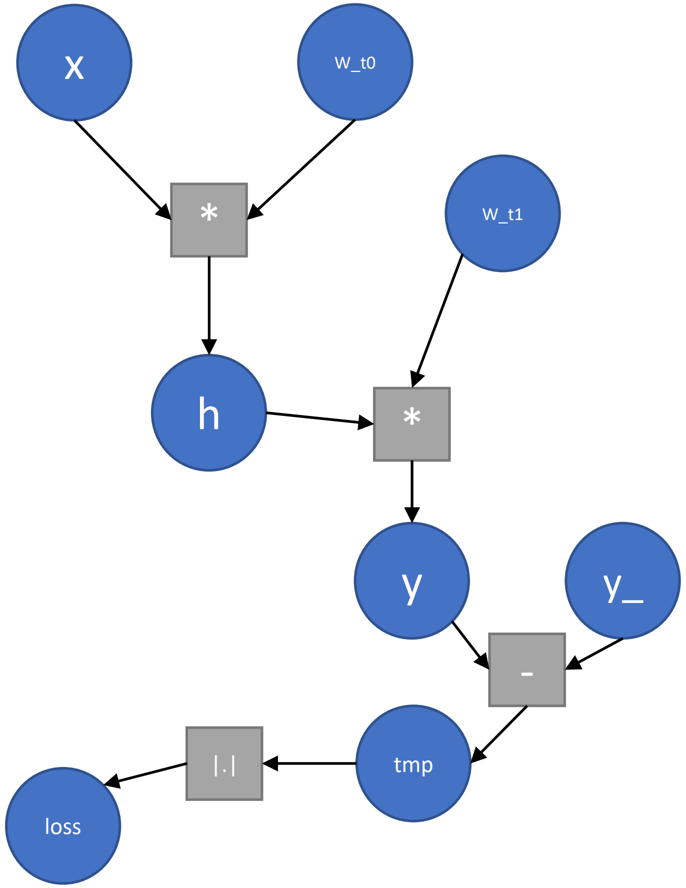
- ```
  x    = np.array([[1.0],[2.1],[3.1]])      #3x1
  W_t0 = tf.Variable(np.random.rand(3,3))   #3x3
  h    = tf.matmul(W_t0, x)                 #3x1

  W_t1 = tf.Variable(np.random.rand(1,3))   #1x3
  y    = tf.matmul(W_t1, x)                 #1x1



  y_   = np.array([[3.3]])
  loss = tf.abs(y-y_)

  log_loss = tf.log(loss)
  ```


  ```

Computational Graph – loss

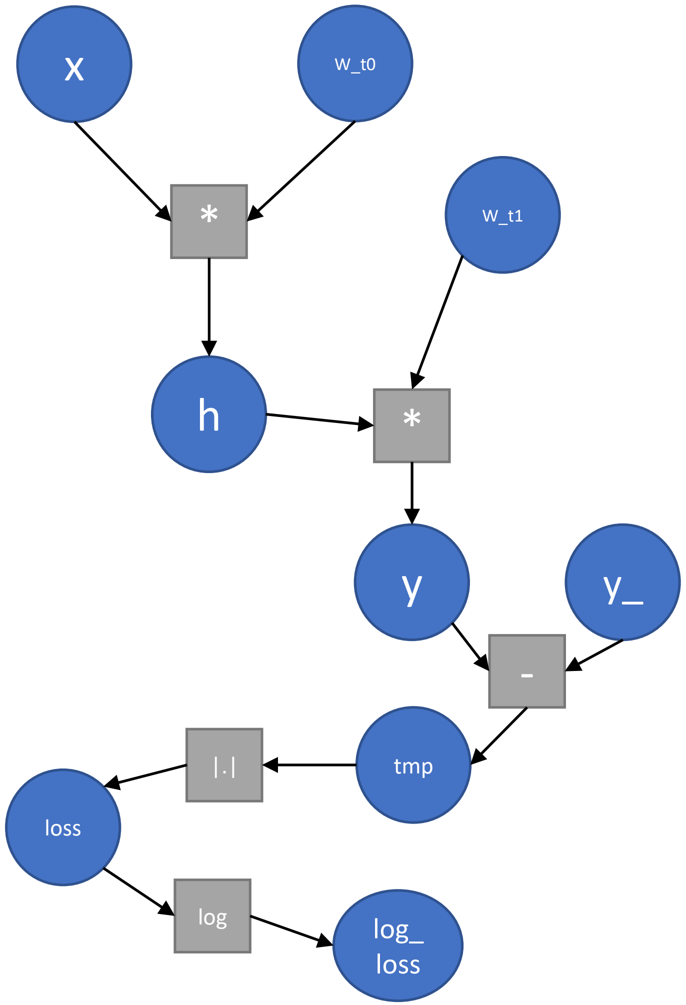
```
x    = np.array([[1.0],[2.1],[3.1]])     #3x1
W_t0 = tf.Variable(np.random.rand(3,3))   #3x3
h    = tf.matmul(W_t0, x)                 #3x1

W_t1 = tf.Variable(np.random.rand(1,3))   #1x3
y    = tf.matmul(W_t1, x)                 #1x1



y_   = np.array([[3.3]])
loss = tf.abs(y-y_)

log_loss = tf.log(loss)
```



Computational Graph – loss

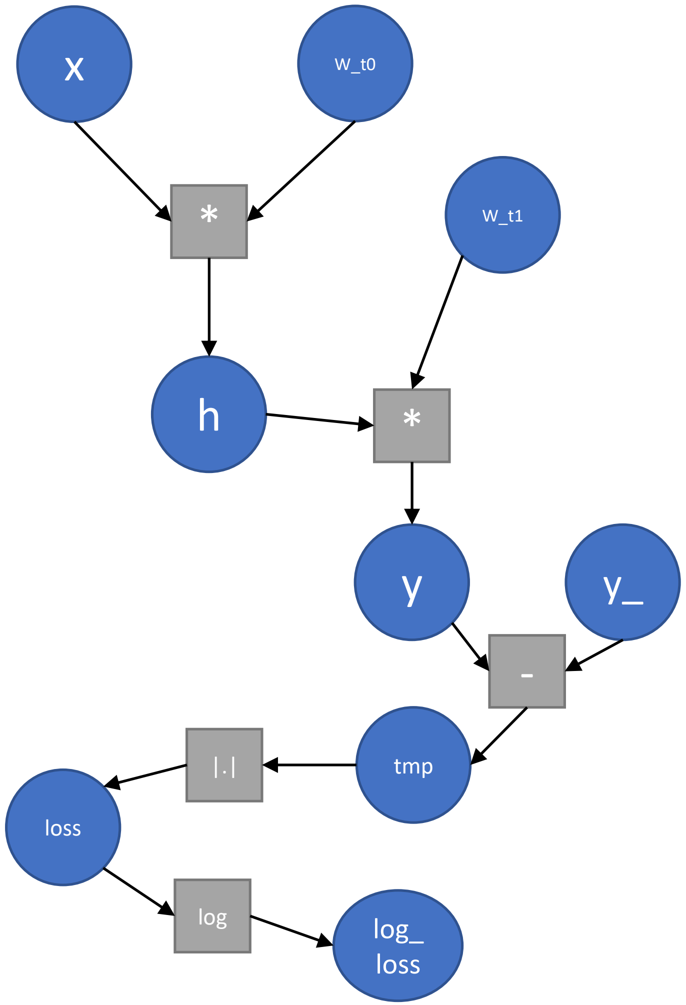
- ```
  x    = np.array([[1.0],[2.1],[3.1]])      #3x1
  W_t0 = tf.Variable(np.random.rand(3,3))   #3x3
  h    = tf.matmul(W_t0, x)                 #3x1

  W_t1 = tf.Variable(np.random.rand(1,3))   #1x3
  y    = tf.matmul(W_t1, x)                 #1x1



  y_   = np.array([[3.3]])
  loss = tf.abs(y-y_)

  log_loss = tf.log(loss)
  ```


  ```

Computational Graph – optimization

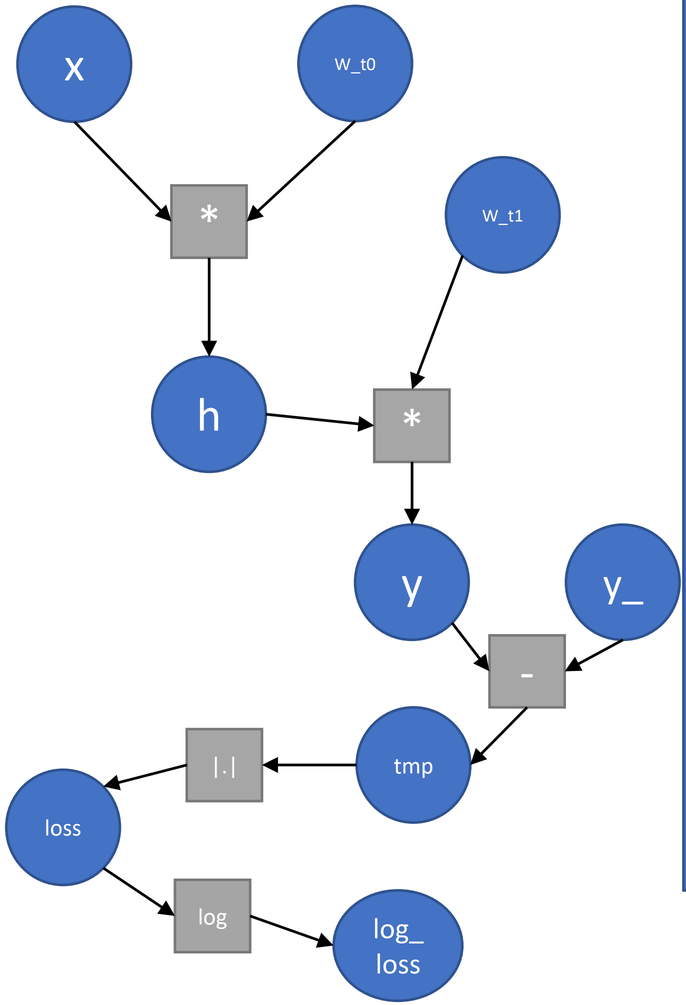
```
• x    = np.array([[1.0],[2.1],[3.1]])      #3x1
  W_t0 = tf.Variable(np.random.rand(3,3))   #3x3
  h    = tf.matmul(W_t0, x)                 #3x1

  W_t1 = tf.Variable(np.random.rand(1,3))   #1x3
  y    = tf.matmul(W_t1, x)                 #1x1



  y_   = np.array([[3.3]])
  loss = tf.abs(y-y_)

  log_loss = tf.log(loss)
```



Goal:

Computational Graph – optimization

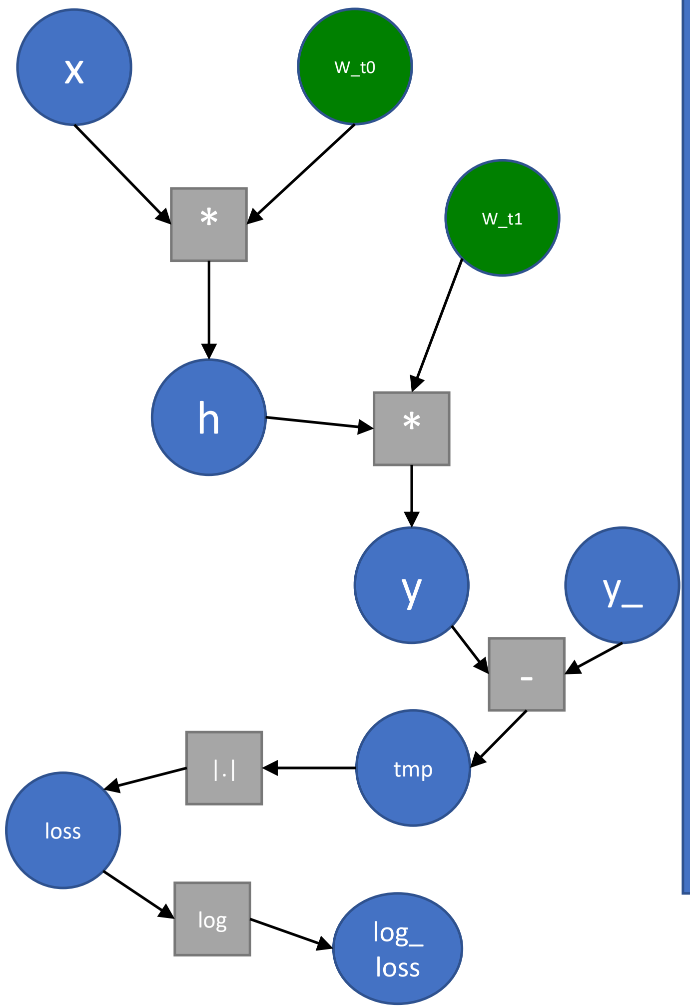
```
x    = np.array([[1.0],[2.1],[3.1]])      #3x1
W_t0 = tf.Variable(np.random.rand(3,3))   #3x3
h    = tf.matmul(W_t0, x)                 #3x1

W_t1 = tf.Variable(np.random.rand(1,3))   #1x3
y    = tf.matmul(W_t1, x)                 #1x1



y_   = np.array([[3.3]])
loss = tf.abs(y-y_)

log_loss = tf.log(loss)
```



Goal:
Adapt variables

Computational Graph – optimization

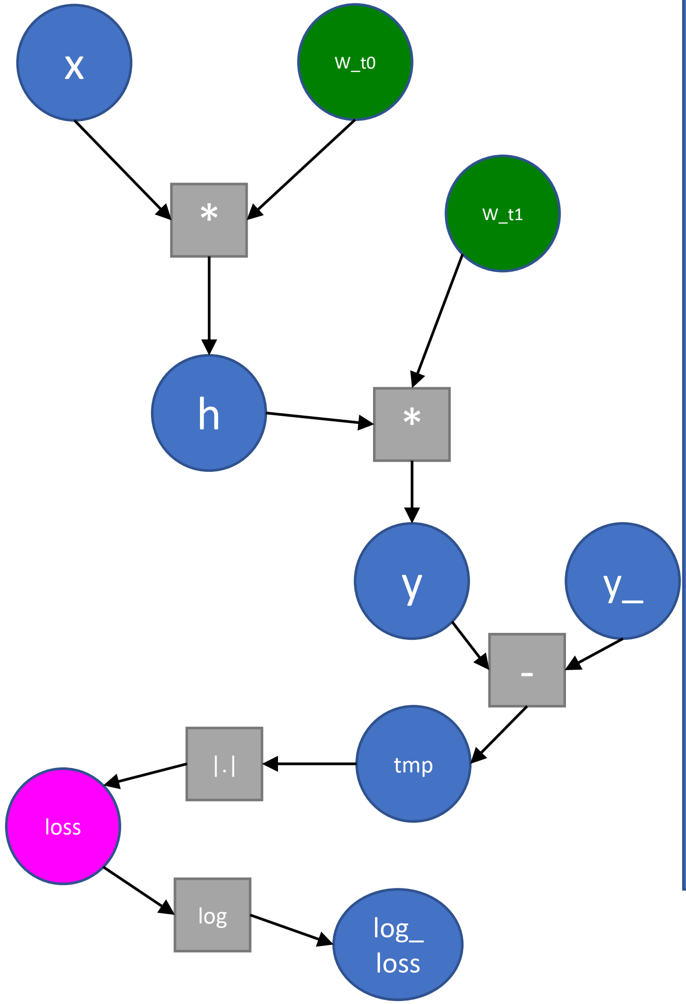
```
x    = np.array([[1.0],[2.1],[3.1]])      #3x1
W_t0 = tf.Variable(np.random.rand(3,3))   #3x3
h    = tf.matmul(W_t0, x)                 #3x1

W_t1 = tf.Variable(np.random.rand(1,3))   #1x3
y    = tf.matmul(W_t1, x)                 #1x1



y_   = np.array([[3.3]])
loss = tf.abs(y-y_)

log_loss = tf.log(loss)
```



Goal:
Adapt variables
to minimize loss!

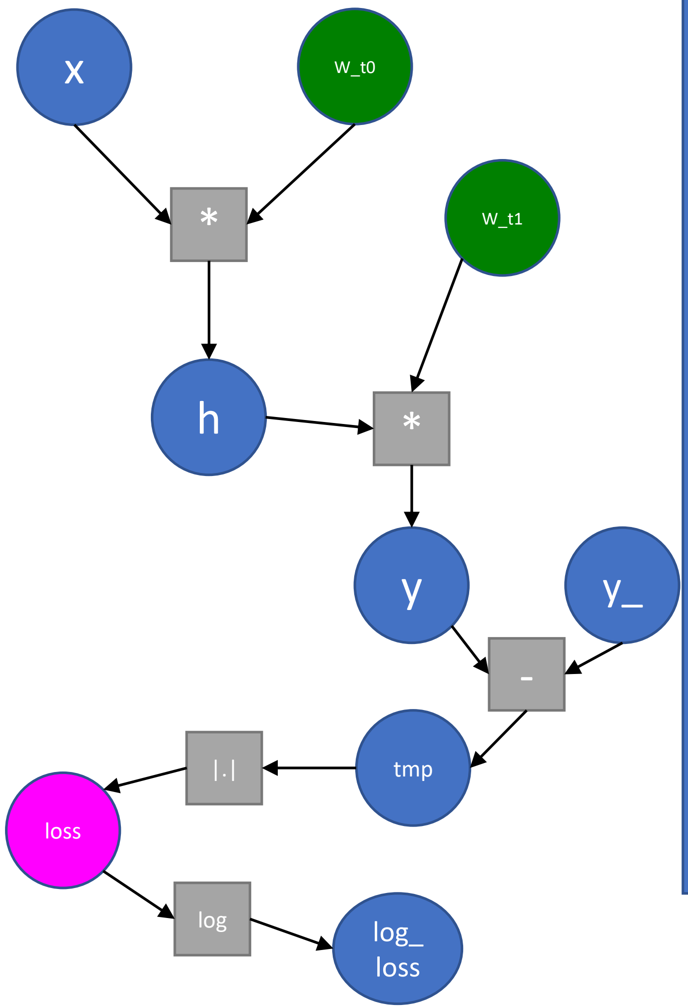
Computational Graph – optimization

```
x    = np.array([[1.0],[2.1],[3.1]])       #3x1
W_t0 = tf.Variable(np.random.rand(3,3))    #3x3
h    = tf.matmul(W_t0, x)                  #3x1

W_t1 = tf.Variable(np.random.rand(1,3))    #1x3
y    = tf.matmul(W_t1, x)                  #1x1


y_   = np.array([[3.3]])
loss = tf.abs(y-y_)

log_loss = tf.log(loss)
```



Goal:
Adapt variables
to minimize loss!

Use gradient based optimizer!

Computational Graph – optimization

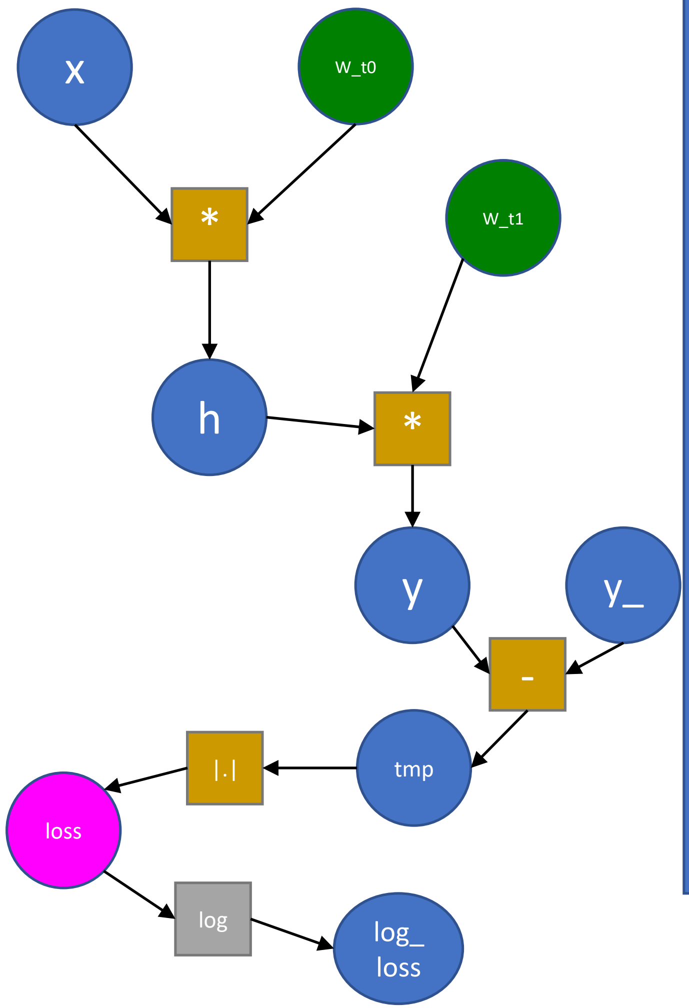
- ```
  x    = np.array([[1.0],[2.1],[3.1]])      #3x1
  W_t0 = tf.Variable(np.random.rand(3,3))   #3x3
  h    = tf.matmul(W_t0, x)                 #3x1

  W_t1 = tf.Variable(np.random.rand(1,3))   #1x3
  y    = tf.matmul(W_t1, x)                 #1x1



  y_   = np.array([[3.3]])
  loss = tf.abs(y-y_)

  log_loss = tf.log(loss)
  ```



Goal:
Adapt variables
to minimize loss!

Use gradient based optimizer!

Need to differentiate along a chain of operations (remember chain rule)!
  ```

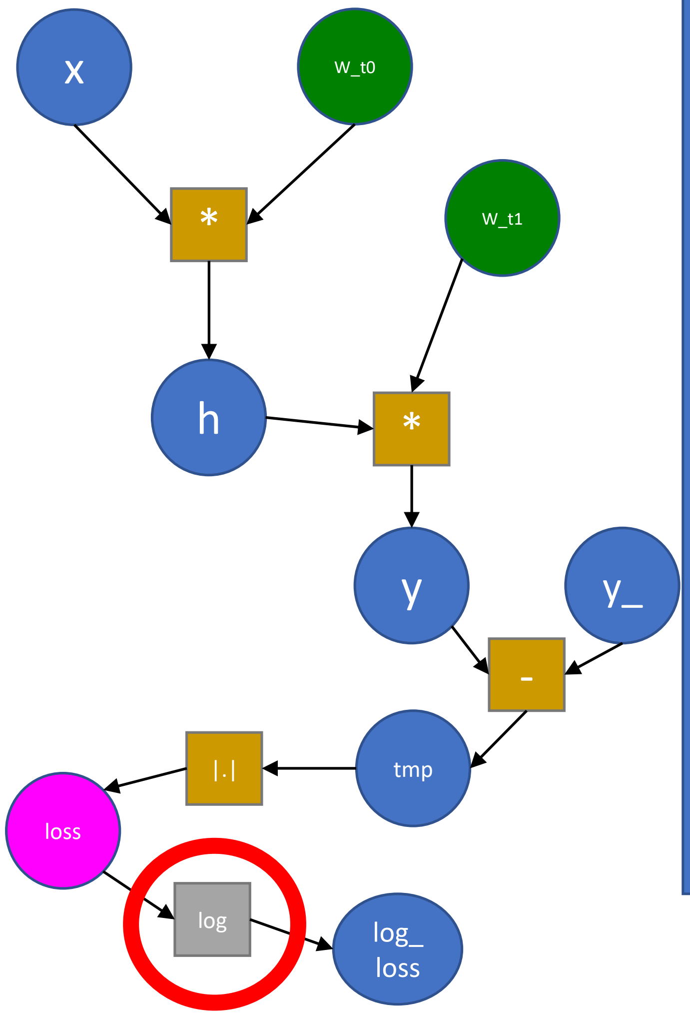
Computational Graph – optimization

```
x    = np.array([[1.0],[2.1],[3.1]])      #3x1
W_t0 = tf.Variable(np.random.rand(3,3))   #3x3
h    = tf.matmul(W_t0, x)                 #3x1

W_t1 = tf.Variable(np.random.rand(1,3))   #1x3
y    = tf.matmul(W_t1, x)                 #1x1


y_   = np.array([[3.3]])
loss = tf.abs(y-y_)

log_loss = tf.log(loss)
```



Goal:
Adapt variables
to minimize loss!

Use gradient based optimizer!

Need to differentiate along a chain of operations (remember chain rule)!

Observation:
Some operations within graph are not needed!

Computational Graph – optimization

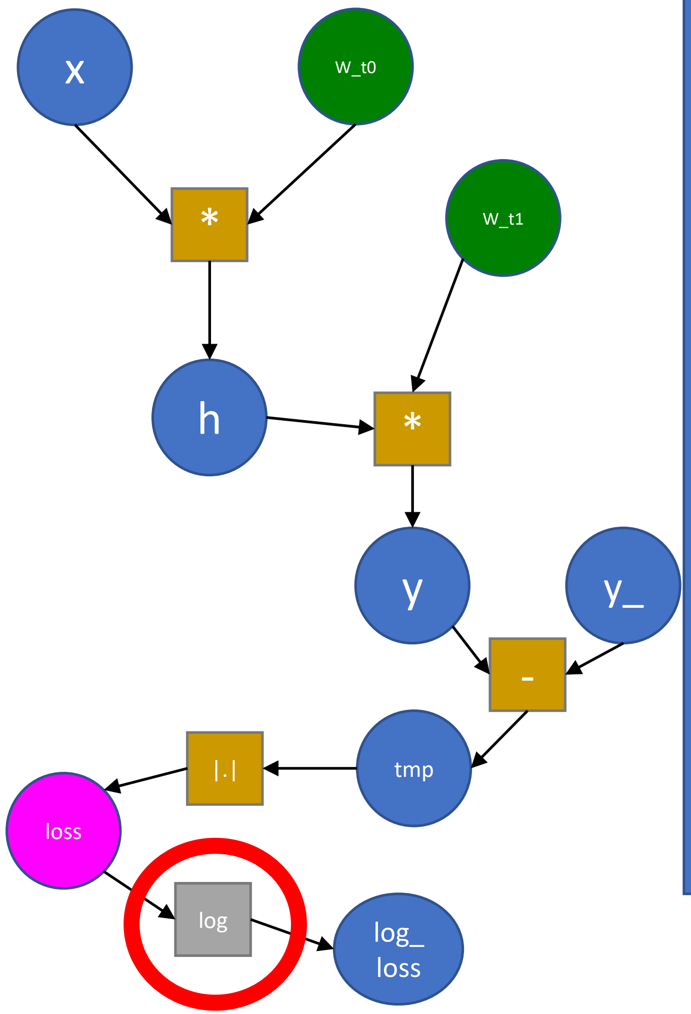
```
x    = np.array([[1.0],[2.1],[3.1]])      #3x1
W_t0 = tf.Variable(np.random.rand(3,3))   #3x3
h    = tf.matmul(W_t0, x)                 #3x1

W_t1 = tf.Variable(np.random.rand(1,3))   #1x3
y    = tf.matmul(W_t1, x)                 #1x1



y_   = np.array([[3.3]])
loss = tf.abs(y-y_)

log_loss = tf.log(loss)
```



Therefore:
„Tape“ only relevant portion of graph for optimization!

Tensorflow - Example

```
import numpy as np
import tensorflow as tf

class myModel():
    def __init__(self, num_inputs, num_outputs):
        self.num_inputs  = num_inputs
        self.weights_t0  = tf.Variable(np.random.rand(3,num_inputs))
        self.weights_t1  = tf.Variable(np.random.rand(num_outputs,3))

        self.optimizer   = tf.keras.optimizers.Adam(learning_rate=0.01)
```

Tensorflow - Example

```
import numpy as np
import tensorflow as tf

class myModel():
    def __init__(self, num_inputs, num_outputs):
        self.num_inputs   = num_inputs
        self.weights_t0   = tf.Variable(np.random.rand(3,num_inputs))
        self.weights_t1   = tf.Variable(np.random.rand(num_outputs,3))

        self.optimizer    = tf.keras.optimizers.Adam(learning_rate=0.01)

    def force_col_vec(self, new_input):
        new_input  = np.array(new_input)
        vec_length = np.prod(new_input.shape)

        return np.reshape(new_input, [vec_length, 1])
```

Tensorflow - Example

```
import numpy as np
import tensorflow as tf

class myModel():
    def __init__(self, num_inputs, num_outputs):
        self.num_inputs   = num_inputs
        self.weights_t0   = tf.Variable(np.random.rand(3,num_inputs))
        self.weights_t1   = tf.Variable(np.random.rand(num_outputs,3))

        self.optimizer    = tf.keras.optimizers.Adam(learning_rate=0.01)

    def force_col_vec(self, new_input):
        new_input   = np.array(new_input)
        vec_length = np.prod(new_input.shape)

        return np.reshape(new_input, [vec_length, 1])

    def get_output(self, new_input):
        new_input = self.force_col_vec(new_input)
        h = tf.matmul(self.weights_t0, new_input)
        y = tf.matmul(self.weights_t1, h)

        return y
```

Tensorflow - Example

```
import numpy as np
import tensorflow as tf

class myModel():
    def __init__(self, num_inputs, num_outputs):
        self.num_inputs  = num_inputs
        self.weights_t0  = tf.Variable(np.random.rand(3,num_inputs))
        self.weights_t1  = tf.Variable(np.random.rand(num_outputs,3))

        self.optimizer   = tf.keras.optimizers.Adam(learning_rate=0.01)

    def force_col_vec(self, new_input):
        new_input  = np.array(new_input)
        vec_length = np.prod(new_input.shape)

        return np.reshape(new_input, [vec_length, 1])

    def get_output(self, new_input):
        new_input = self.force_col_vec(new_input)
        h = tf.matmul(self.weights_t0, new_input)
        y = tf.matmul(self.weights_t1, h)

        return y

    def get_loss(self, new_input, target_output):
        model_output = self.get_output(new_input)
        loss = tf.abs(model_output- target_output)

        return loss
```

Tensorflow - Example

```
import numpy as np
import tensorflow as tf

class myModel():
    def __init__(self, num_inputs, num_outputs):
        self.num_inputs  = num_inputs
        self.weights_t0  = tf.Variable(np.random.rand(3,num_inputs))
        self.weights_t1  = tf.Variable(np.random.rand(num_outputs,3))

        self.optimizer   = tf.keras.optimizers.Adam(learning_rate=0.01)

    def force_col_vec(self, new_input):
        new_input  = np.array(new_input)
        vec_length = np.prod(new_input.shape)

        return np.reshape(new_input, [vec_length, 1])

    def get_output(self, new_input):
        new_input = self.force_col_vec(new_input)
        h = tf.matmul(self.weights_t0, new_input)
        y = tf.matmul(self.weights_t1, h)

        return y

    def get_loss(self, new_input, target_output):
        model_output = self.get_output(new_input)
        loss = tf.abs(model_output- target_output)

        return loss

    def _get_gradient(self, new_input, target_output):
        with tf.GradientTape() as tape:
            loss = self.get_loss(new_input, target_output)
            grad = tape.gradient(loss,[self. weights_t0, self.weights_t1])

        return grad
```

Tensorflow - Example

```
import numpy as np
import tensorflow as tf

class myModel():
    def __init__(self, num_inputs, num_outputs):
        self.num_inputs  = num_inputs
        self.weights_t0  = tf.Variable(np.random.rand(3,num_inputs))
        self.weights_t1  = tf.Variable(np.random.rand(num_outputs,3))

        self.optimizer   = tf.keras.optimizers.Adam(learning_rate=0.01)

    def force_col_vec(self, new_input):
        new_input  = np.array(new_input)
        vec_length = np.prod(new_input.shape)

        return np.reshape(new_input, [vec_length, 1])

    def get_output(self, new_input):
        new_input = self.force_col_vec(new_input)
        h = tf.matmul(self.weights_t0, new_input)
        y = tf.matmul(self.weights_t1, h)

        return y

    def get_loss(self, new_input, target_output):
        model_output = self.get_output(new_input)
        loss = tf.abs(model_output- target_output)

        return loss

    def _get_gradient(self, new_input, target_output):
        with tf.GradientTape() as tape:
            loss = self.get_loss(new_input, target_output)
            grad = tape.gradient(loss,[self. weights_t0, self.weights_t1])

        return grad

    def update_weights(self, new_input, target_output):
        grad = self._get_gradient(new_input, target_output)
        self.optimizer.apply_gradients(zip(grad ,[self. weights_t0, self.weights_t1]))
```

Tensorflow - Example

```
import numpy as np
import tensorflow as tf

class myModel():
    def __init__(self, num_inputs, num_outputs):
        self.num_inputs  = num_inputs
        self.weights_t0  = tf.Variable(np.random.rand(3,num_inputs))
        self.weights_t1  = tf.Variable(np.random.rand(num_outputs,3))

        self.optimizer   = tf.keras.optimizers.Adam(learning_rate=0.01)

    def force_col_vec(self, new_input):
        new_input  = np.array(new_input)
        vec_length = np.prod(new_input.shape)

        return np.reshape(new_input, [vec_length, 1])

    def get_output(self, new_input):
        new_input = self.force_col_vec(new_input)
        h = tf.matmul(self.weights_t0, new_input)
        y = tf.matmul(self.weights_t1, h)

        return y

    def get_loss(self, new_input, target_output):
        model_output = self.get_output(new_input)
        loss = tf.abs(model_output- target_output)

        return loss

    def _get_gradient(self, new_input, target_output):
        with tf.GradientTape() as tape:
            loss = self.get_loss(new_input, target_output)
            grad = tape.gradient(loss,[self. weights_t0, self.weights_t1])

        return grad

    def update_weights(self, new_input, target_output):
        grad = self._get_gradient(new_input, target_output)
        self.optimizer.apply_gradients(zip(grad ,[self. weights_t0, self.weights_t1]))
```

Content

- Revision: Practical Task
- Revision: Lecture
- Tensorflow
- New Practical Task