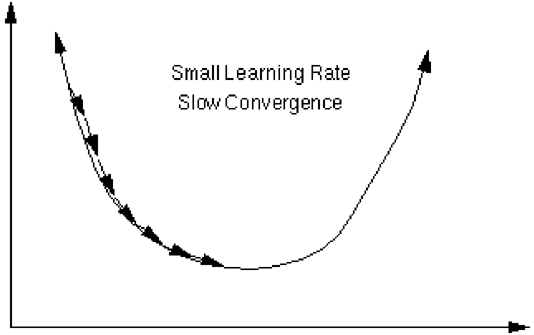
工業技術研究院
Industrial Technology
Research Institute

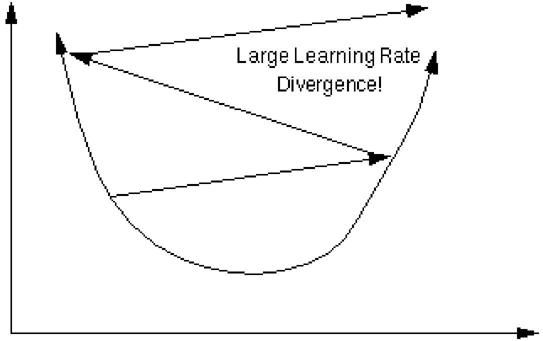
Basic Neural Net Training using Python

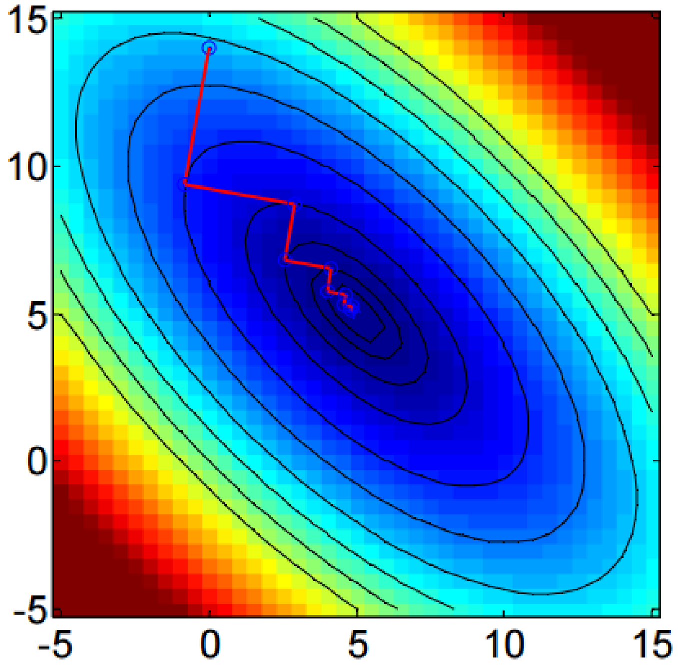
Alex Lin

Safety Sensing & Control Department
Intelligent Mobility Technology Division
Mechanical and Systems Research Laboratories
Industrial Technology Research Institute

How Gradient Descent actually works?



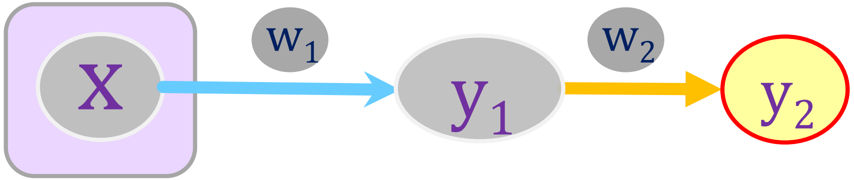




Goal

- In the following examples, you will have learnt
  - How numpy array is helpful in doing forward/backward pass
  - Why Python is modular, high level….etc.,
  - How gradient flow is related to back-propagation
  - How Neural Nets actually work
  - How Relu works and when it is dead
  - How back-propagation is actually done with and without mini-batch.

Two layers with 1 input and 1 output



- There is a relu in y_1
- $y_1=w_1x$ and $y_2=w_2y_1$
- In the learning process, both w_1 and w_2 are adjusted in hope that y_2 approaches its ground-truth $\overline{y}_2$.
- Here, we adopt 2nd norm for the loss function.
- Loss= $(\overline{y}_2 - y_2)^2$.
- By chain rule, $\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial y_2}\frac{\partial y_2}{\partial w_2} = 2(\overline{y}_2 - y_2)\ y_1$, $\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y_2}\frac{\partial y_2}{\partial y_1}\frac{\partial y_1}{\partial w_1} = 2(\overline{y}_2 - y_2)\ w_2x$
- $w_2=w_2-\alpha\frac{\partial L}{\partial w_2}$, $w_1=w_1-\alpha\frac{\partial L}{\partial w_1}$ where α = learning rate

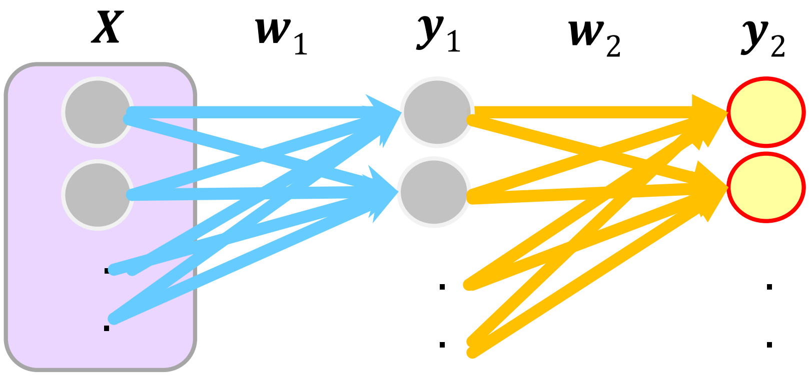
Python code

```
for t in range(iterations):
    # 1st layer inference
    y1 = x.dot(w1)
    # doing relu for the output of 1st layer
    y1_relu = np.maximum(y1, 0)
    # 2nd layer inference
    y2_pred = y1_relu.dot(w2)
    # output of the whole neural net
    Inference_result_history[t]=y2_pred
    # Compute the loss
    loss = np.square(y2_pred - y2_GT).sum()
     # Backprop to compute gradients of w1 and w2 with respect to loss
    grad_y2_pred = 2.0 * (y2_pred - y2_GT) # d_loss/d_y2
    grad_w2 = y1_relu.dot(grad_y2_pred) # (d_loss/d_y2)*(d_y2/d_w2)=d_loss/d_w2
    grad_y1_relu = grad_y2_pred.dot(w2) # (d_loss/d_y2)*(d_y2/d_y1)=d_loss/d_y1
    grad_y1 = grad_y1_relu.copy()
    grad_y1[y1 < 0] = 0 # only weightings through relu would be conducted back pass
    grad_w1 = x.dot(grad_y1) # (d_loss/d_y2)*(d_y2/d_y1)*(d_y1/d_w1)=(d_loss/d_y1)*(d_y1/d_w1)=d_loss/d_w1
    # Update weights
    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

Discussion

- In what situation would this neural net fail?

Two layers with multiple-dimensional input and output



- Every neuron in n $\boldsymbol{y}_1$ and $\boldsymbol{y}_2$ accompanies a relu
- $\boldsymbol{x}$ (1-by-k), $\boldsymbol{y}_1$ (1-by-n) and $\boldsymbol{y}_2$ (1-by-m) are vectors; $\boldsymbol{w}_1$ (k-by-n) & $\boldsymbol{w}_2$ (n-by-m) are matrices.
- $\boldsymbol{y}_1 = \boldsymbol{x}\boldsymbol{w}_1$ and $\boldsymbol{y}_2 = \boldsymbol{y}_1\boldsymbol{w}_2$
- In the learning process, both $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$ are adjusted in hope that $\mathbf{y}_2$ approaches its ground-truth $\bar{\mathbf{y}}_2$.
- Here, we adopt 2nd norm for the loss function.
- Loss= $(\overline{\boldsymbol{y}_2} - \boldsymbol{y}_2)^2$.
- By chain rule, $\frac{\partial L}{\partial \boldsymbol{w}_2} = \frac{\partial \boldsymbol{y}_2}{\partial \boldsymbol{w}_2}\frac{\partial L}{\partial \boldsymbol{y}_2} = 2\boldsymbol{y}_1^t(\overline{\boldsymbol{y}_2} - \boldsymbol{y}_2)$, $\frac{\partial L}{\partial \boldsymbol{w}_1} = \frac{\partial \boldsymbol{y}_1}{\partial \boldsymbol{w}_1}\frac{\partial L}{\partial \boldsymbol{y}_2}\frac{\partial \boldsymbol{y}_2}{\partial \boldsymbol{y}_1} = 2\boldsymbol{x}^t(\overline{\boldsymbol{y}_2} - \boldsymbol{y}_2)\,\boldsymbol{w}_2^t$
- $\boldsymbol{w}_2 = \boldsymbol{w}_2 - \alpha\frac{\partial L}{\partial \boldsymbol{w}_2}$, $\boldsymbol{w}_1 = \boldsymbol{w}_1 - \alpha\frac{\partial L}{\partial \boldsymbol{w}_1}$ where α = learning rate

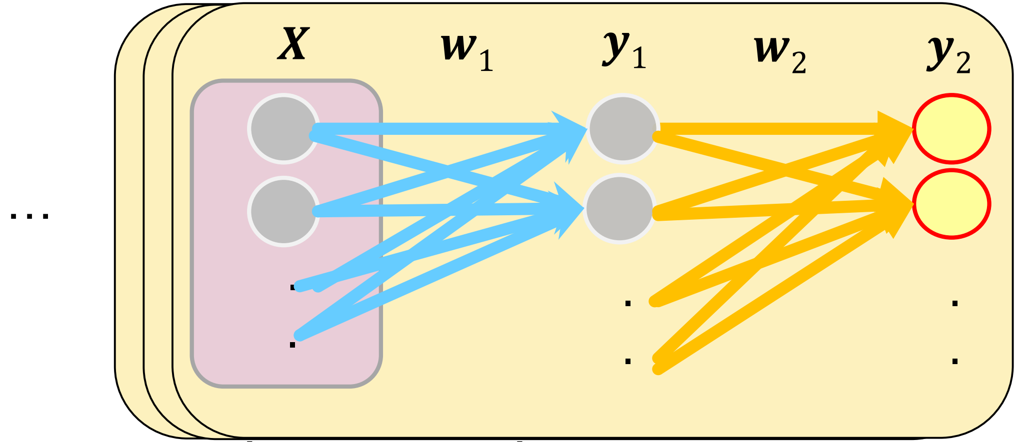
Python code

```
for t in range(iterations):
    # 1st layer inference
    y1 = x.dot(w1)
        # doing relu for the output of 1st layer
    y1_relu = np.maximum(y1, 0) # result is a row vector
    # store the output of the 1st layer
    y1_history[t]= np.mean(y1_relu)
    # performing 2nd layer computation
    y2_pred = y1_relu.dot(w2) # result is a row vector
    # Compute and print loss
    loss = np.square(y2_pred - y).sum()
    # Backprop to compute gradients of w1 and w2 with respect to loss
    grad_y2_pred = 2.0 * (y2_pred - y) # d_loss/d_y2
    grad_w2 = y1_relu.T.dot(grad_y2_pred)# (d_y2/d_w2)*(d_loss/d_y2)=d_loss/d_w2
    grad_y1_relu = grad_y2_pred.dot(w2.T) # (d_loss/d_y2)*(d_y2/d_y1)=d_loss/d_y1
    grad_y1 = grad_y1_relu.copy()
    grad_y1[y1 < 0] = 0 # only numbers through relu would be conducted backward pass
    grad_w1 = x.T.dot(grad_y1) # (d_y1/d_w1)*(d_loss/d_y2)*(d_y2/d_y1)=(d_y1/d_w1)*(d_loss/d_y1)=d_loss/d_w1
    # Update weights
    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

Discussion

- Why is the appropriate learning rate in comparison to case 1 in terms of the number of inputs and outputs?

Two layers with multiple-dimensional inputs and outputs (mini-batch)



- Every neuron in n $\boldsymbol{y}_1$ and $\boldsymbol{y}_2$ accompanies a relu
- $\boldsymbol{x}$ (N-by-k), $\boldsymbol{y}_1$ (N-by-n) and $\boldsymbol{y}_2$ (N-by-m) are vectors; $\boldsymbol{w}_1$ (k-by-n) & $\boldsymbol{w}_2$ (n-by-m) are matrices.
- $\boldsymbol{y}_1 = \boldsymbol{x}\boldsymbol{w}_1$ and $\boldsymbol{y}_2 = \boldsymbol{y}_1\boldsymbol{w}_2$
- In the learning process, both $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$ are adjusted in hope that $\mathbf{y}_2$ approaches its ground-truth $\bar{\mathbf{y}}_2$.
- Here, we adopt 2nd norm for the loss function.
- Loss= $(\bar{y}_2 - y_2)^2$.
- By chain rule, $\frac{\partial L}{\partial \boldsymbol{w}_2} = \frac{\partial \boldsymbol{y}_2}{\partial \boldsymbol{w}_2}\frac{\partial L}{\partial \boldsymbol{y}_2} = 2\boldsymbol{y}_1^t(\bar{\boldsymbol{y}}_2 - \boldsymbol{y}_2)$, $\frac{\partial L}{\partial \boldsymbol{w}_1} = \frac{\partial \boldsymbol{y}_1}{\partial \boldsymbol{w}_1}\frac{\partial L}{\partial \boldsymbol{y}_2}\frac{\partial \boldsymbol{y}_2}{\partial \boldsymbol{y}_1} = 2\boldsymbol{x}^t(\bar{\boldsymbol{y}}_2 - \boldsymbol{y}_2)\,\boldsymbol{w}_2^t$
- $\boldsymbol{w}_2 = \boldsymbol{w}_2 - \alpha\frac{\partial L}{\partial \boldsymbol{w}_2}$, $\boldsymbol{w}_1 = \boldsymbol{w}_1 - \alpha\frac{\partial L}{\partial \boldsymbol{w}_1}$ where α = learning rate

Python code

```
for t in range(iterations):
    # 1st layer inference
    y1 = x.dot(w1)
        # doing relu for the output of 1st layer
    y1_relu = np.maximum(y1, 0) # result is a matrix
    # store the output of the 1st layer
    y1_history[t]= np.mean(y1_relu)
    # performing 2nd layer computation
    y2_pred = y1_relu.dot(w2) # result is a row vector
    # Compute and print loss
    loss = np.square(y2_pred - y2_GT).sum()
    # Backprop to compute gradients of w1 and w2 with respect to loss
    grad_y2_pred = 2.0 * (y2_pred - y2_GT) # d_loss/d_y2
    grad_w2 = y1_relu.T.dot(grad_y2_pred)# (d_y2/d_w2)*(d_loss/d_y2)=d_loss/d_w2
    grad_y1_relu = grad_y2_pred.dot(w2.T) # (d_loss/d_y2)*(d_y2/d_y1)=d_loss/d_y1
    grad_y1 = grad_y1_relu.copy()
    grad_y1[y1 < 0] = 0 # only numbers through relu would be conducted backward pass
    grad_w1 = x.T.dot(grad_y1) # (d_y1/d_w1)*(d_loss/d_y2)*(d_y2/d_y1)=(d_y1/d_w1)*(d_loss/d_y1)=d_loss/d_w1
    # Update weights
    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

Discussion

- What is gradient explosion?
- How can you manually produce dead Relu or gradient explosion in terms of hyperparameters?

Thank you!