FIT3181/5215 Deep Learning

Week 04: Back-Propagation and Optimization for Deep Learning

Lecturers: Dr Trung Le

A/Prof Zongyuan Ge

Dr Arghya Pal

Email: trunglm@monash.edu

GROUP OF EIGHT AUSTRALIA

Department of Data Science and AI

Faculty of Information Technology, Monash University, Australia

Outline

- Revision of calculus.
- Gradient descent and stochastic gradient descent.
- Backpropagation in feed-forward neural networks.
- Optimizers for deep learning.

- **Further reading recommendations**
  - Deep Learning – Chapter 8
  - Dive into Deep Learning – Chapter 11 (https://d2l.ai/chapter_optimization/index.html)
  - Ruder's blog: https://ruder.io/optimizing-gradient-descent/index.html

Our Story

```python
class OurFFN(torch.nn.Module):
  def __init__(self, n_features, n_classes):
    super(OurFFN, self).__init__()
    self.n_features = n_features
    self.n_classes = n_classes
    self.fc1 = nn.Linear(n_features, 10)
    self.fc2 = nn.Linear(10, 20)
    self.fc3 = nn.Linear(20, 15)
    self.fc4 = nn.Linear(15, n_classes)

  def forward(self, x): #x is the mini-batch
    x = self.fc1(x)
    x = F.relu(x)
    x = self.fc2(x)
    x = F.relu(x)
    x = self.fc3(x)
    x = F.relu(x)
    x = self.fc4(x)
    return x
```

Declare the model

```python
# Loss and optimizer
learning_rate = 0.005
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(ffn_model.parameters(), lr=learning_rate)
```

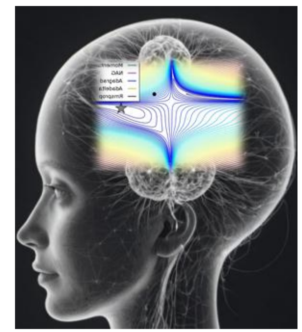
Declare loss and optimizer

```python
for epoch in range(num_epochs):
    for i, (X, y) in enumerate(train_loader):
        X, y = X.to(device), y.to(device)
        # Forward pass
        outputs = ffn_model(X.type(torch.float32))
        loss = loss_fn(outputs, y.type(torch.long))
        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

Train the model

Back-propagation

Optimizers

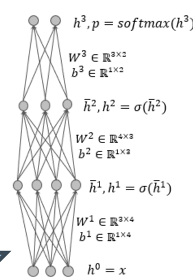


loss.backward()

optimizer.step()

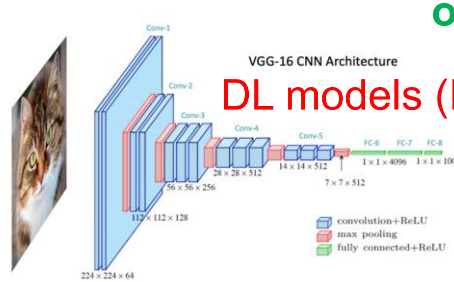
1. **Small detour to calculus**
2. **Loss landscape of DL models**
3. **Back-propagation**
4. **Optimizers for DL**

Backprop



Compute gradients

$$\frac{\partial l}{\partial W}, \frac{\partial l}{\partial b}$$



DL models (FFN, CNN, RNN, ViT,...)

Update

$$W = W - \eta \frac{\partial l}{\partial W}$$

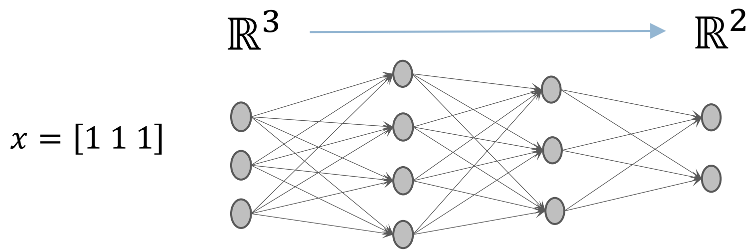
$$b = b - \eta \frac{\partial l}{\partial b}$$

Small Detour to Calculus

A small detour to calculus

- Calculus = **mathematics of change** (very important for deep learning)
- Properties of derivative:
  - $f'(x) = \nabla f(x) = \lim_{h \to 0} \frac{f(x+h)-f(x)}{h}$
  - $(uv)' = u'v + uv'$
  - $\left(\frac{u}{v}\right)' = \frac{u'v-uv'}{v^2}$
  - $(e^u)' = u'e^u$
  - $(\log u)' = \frac{u'}{u}$
- Multi-variate function $f\colon \mathbb{R}^n \to \mathbb{R}$ with $y = f(x) = f(x_1, \dots, x_n)$.
  - Gradient/derivative: $\frac{\partial f}{\partial x}(a) = \nabla_x f(a) = [\nabla_{x_1} f(a), \nabla_{x_2} f(a), \dots, \nabla_{x_n} f(a)]$.
- Chain rule 🔗 :
  - $\frac{\partial u}{\partial x} = \frac{\partial u}{\partial v} \times \frac{\partial v}{\partial x}$

Derivative for multi-variate functions



- Given a function $f: \mathbb{R}^m \to \mathbb{R}^n$
  - $f(x) = (f_1(x), \dots, f_n(x))$ where $f_1, \dots, f_n: \mathbb{R}^m \to \mathbb{R}$ and $x = (x_1, \dots, x_m)$. Let denote $y = f(x)$.
  - The **derivative** of f at the **point** $\boldsymbol{a} \in \mathbb{R}^m$, denoted by $\nabla f(a)$ (**function related** notion) or $\frac{\partial y}{\partial x}(a)$ (**variable related** notion) is a matrix n by m (i.e., the Jacobian matrix).

$$\frac{\partial y}{\partial x}(a) = \nabla f(a) = \begin{array}{c} \begin{array}{cccccc} & & & m & & \end{array} \\ \begin{bmatrix} \frac{\partial f_1}{\partial x_1}(a) & \dots & \dots & \frac{\partial f_1}{\partial x_j}(a) & \dots & \frac{\partial f_1}{\partial x_m}(a) \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \frac{\partial f_i}{\partial x_1}(a) & \dots & \dots & \frac{\partial f_i}{\partial x_j}(a) & \dots & \frac{\partial f_i}{\partial x_m}(a) \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \frac{\partial f_n}{\partial x_1}(a) & \dots & \dots & \frac{\partial f_n}{\partial x_j}(a) & \dots & \frac{\partial f_n}{\partial x_m}(a) \end{bmatrix} n \end{array}$$

Jacobian matrix

- **Chain rule** 🔗
  - Given a function $\boldsymbol{f}: \mathbb{R}^{\boldsymbol{m}} \to \mathbb{R}^{\boldsymbol{n}}$, $\boldsymbol{g}: \mathbb{R}^{\boldsymbol{n}} \to \mathbb{R}^{\boldsymbol{p}}$, denote $\boldsymbol{h} = \boldsymbol{g} \circ \boldsymbol{f}: \mathbb{R}^{\boldsymbol{m}} \to \mathbb{R}^{\boldsymbol{p}}$, meaning that $\boldsymbol{h}(\boldsymbol{x}) = \boldsymbol{g}(\boldsymbol{f}(\boldsymbol{x}))$. We further define $y = f(x)$ and $z = g(y) = g(f(x)) = h(x)$.
  - For $x \in \mathbb{R}^m$, $\underset{p \times m}{\nabla h(x)} = \underset{p \times n}{\nabla g(f(x))} \times \underset{n \times m}{\nabla f(x)}$ or equivalently $\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} . \frac{\partial y}{\partial x}$.

$x \qquad y = f(x) \qquad z = g(y) = g(f(x)) = h(x)$

$$\begin{tikzcd} \mathbb{R}^m \arrow[r, "f"] \arrow[rr, bend right, "h"'] & \mathbb{R}^n \arrow[r, "g"] & \mathbb{R}^p \end{tikzcd}$$

Example

- $y = f(x) = f(x_1, x_2, x_3) = (x_1^2 + x_2^2, x_2^2 + x_3^2 x_2)$
  - $f : \mathbb{R}^3 \to \mathbb{R}^2$
  - $f_1(x) = f_1(x_1, x_2, x_3) = x_1^2 + x_2^2$
  - $f_2(x) = f_2(x_1, x_2, x_3) = x_2^2 + x_3^2 x_2$
  - $\frac{\partial y}{\partial x} = \nabla f \in \mathbb{R}^{2\times 3}$

$$\frac{\partial y}{\partial x} = \nabla_x f = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \frac{\partial f_1}{\partial x_3} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \frac{\partial f_2}{\partial x_3} \end{bmatrix} = \begin{bmatrix} 2x_1 & 2x_2 & 0 \\ 0 & 2x_2 + x_3^2 & 2x_2 x_3 \end{bmatrix}$$

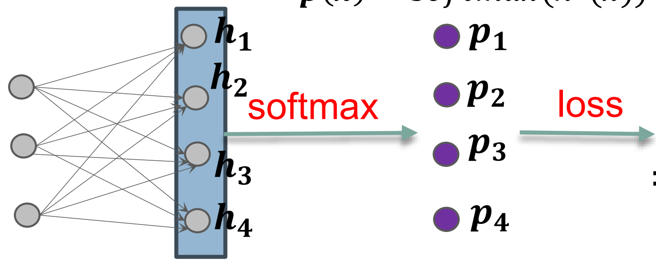
Example

Output layer

$x = [x_1\ x_2\ x_2] \in \mathbb{R}^{1\times 3}$
$y = 2$

Penultimate Layer **Output** Layer **Prediction Probabilities**

$\boldsymbol{p}(\boldsymbol{x}) = softmax(h^L(x))$



Logit $h = [h_1\ h_2\ h_3\ h_4]$ Prob p $= [p_1\ p_2\ p_3\ p_4]$

$$l = loss = -\log p_2$$
$$= -\,log\,\frac{e^{h_2}}{e^{h_1}+e^{h_2}+e^{h_3}+e^{h_4}}$$

$$\boldsymbol{p_1} = \frac{\boldsymbol{e^{h_1}}}{\boldsymbol{e^{h_1} + e^{h_2} + e^{h_3} + e^{h_4}}}$$
$$\boldsymbol{p_2} = \frac{\boldsymbol{e^{h_2}}}{\boldsymbol{e^{h_1} + e^{h_2} + e^{h_3} + e^{h_4}}}$$
$$\boldsymbol{p_3} = \frac{\boldsymbol{e^{h_3}}}{\boldsymbol{e^{h_1} + e^{h_2} + e^{h_3} + e^{h_4}}}$$
$$\boldsymbol{p_4} = \frac{\boldsymbol{e^{h_4}}}{\boldsymbol{e^{h_1} + e^{h_2} + e^{h_3} + e^{h_4}}}$$

Compute $\frac{\partial l}{\partial \boldsymbol{h}}$?

- $l = -\,log\,\frac{e^{h_2}}{e^{h_1}+e^{h_2}+e^{h_3}+e^{h_4}} = \log\big(\overbrace{e^{h_1} + e^{h_2} + e^{h_3} + e^{h_4}}^{u}\big) - h_2$
- $\frac{\partial l}{\partial h_1} = \frac{\nabla_{h_1} u}{u} = \frac{e^{h_1}}{e^{h_1}+e^{h_2}+e^{h_3}+e^{h_4}} = p_1$
- $\frac{\partial l}{\partial h_2} = \frac{\nabla_{h_2} u}{u} - 1 = \frac{e^{h_2}}{e^{h_1}+e^{h_2}+e^{h_3}+e^{h_4}} - 1 = p_2 - 1$
- $\frac{\partial l}{\partial h_3} = \frac{\nabla_{h_3} u}{u} = \frac{e^{h_3}}{e^{h_1}+e^{h_2}+e^{h_3}+e^{h_4}} = p_3$
- $\frac{\partial l}{\partial h_4} = \frac{\nabla_{h_4} u}{u} = \frac{e^{h_4}}{e^{h_1}+e^{h_2}+e^{h_3}+e^{h_4}} = p_4$
- $\frac{\partial l}{\partial \boldsymbol{h}} = [p_1, p_2 - 1, p_3, p_4] = [p_1, p_2, p_3, p_4] - [0,1,0,0] = \boldsymbol{p} - \boldsymbol{1_2} = \boldsymbol{p} - \boldsymbol{1_y}$

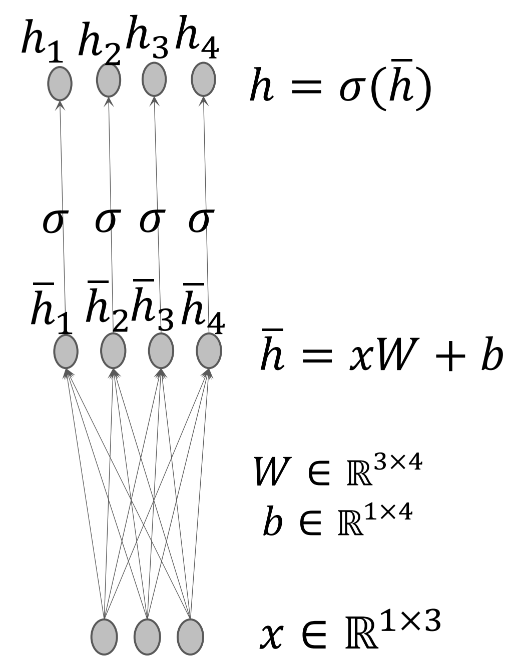
Example

Intermediate layer

- $\bar{h} = xW + b$ and $h = \sigma(\bar{h})$
  - $h = \sigma(xW + b)$
  - σ is the **activation function**
- $\dfrac{\partial h}{\partial x} = \dfrac{\partial h}{\partial \bar{h}} \times \dfrac{\partial \bar{h}}{\partial x} = diag\left(\sigma'(\bar{h})\right) W^T \in \mathbb{R}^{4\times 3}$

- $$\frac{\partial h}{\partial \bar{h}} = \begin{bmatrix} \frac{\partial h_1}{\partial \bar{h}_1} & \frac{\partial h_1}{\partial \bar{h}_2} & \frac{\partial h_1}{\partial \bar{h}_3} & \frac{\partial h_1}{\partial \bar{h}_4} \\ \frac{\partial h_2}{\partial \bar{h}_1} & \frac{\partial h_2}{\partial \bar{h}_2} & \frac{\partial h_2}{\partial \bar{h}_3} & \frac{\partial h_2}{\partial \bar{h}_4} \\ \frac{\partial h_3}{\partial \bar{h}_1} & \frac{\partial h_3}{\partial \bar{h}_2} & \frac{\partial h_3}{\partial \bar{h}_3} & \frac{\partial h_3}{\partial \bar{h}_4} \\ \frac{\partial h_4}{\partial \bar{h}_1} & \frac{\partial h_4}{\partial \bar{h}_2} & \frac{\partial h_4}{\partial \bar{h}_3} & \frac{\partial h_4}{\partial \bar{h}_4} \end{bmatrix} = \begin{bmatrix} \sigma'(\bar{h}_1) & 0 & 0 & 0 \\ 0 & \sigma'(\bar{h}_2) & 0 & 0 \\ 0 & 0 & \sigma'(\bar{h}_3) & 0 \\ 0 & 0 & 0 & \sigma'(\bar{h}_4) \end{bmatrix} = diag(\sigma'(\bar{h}))$$

- $\dfrac{\partial \bar{h}}{\partial x} = W^T$



How to code with PyTorch

- $\bar{h} = xW + b$ and $h = sigmoid(\bar{h})$
  - $h = sigmoid(xW + b)$
  - $\sigma = sigmoid$ is the **activation function**
- $\frac{\partial h}{\partial x} = \frac{\partial h}{\partial \bar{h}} \times \frac{\partial \bar{h}}{\partial x} = diag\left(\sigma'(\bar{h})\right) W^T = diag(\sigma(\bar{h}) \otimes [1 - \sigma(\bar{h})]) W^T = diag(h \otimes (1 - h)) W^T$
  - $\otimes$ is element-wise product

```
x = torch.tensor([1,-1,1], dtype=torch.float32)
print(x)
W = torch.rand(3,4)
b = torch.rand(1,4)
print(W)
print(b)
```

```
tensor([ 1., -1.,  1.])
tensor([[0.7266, 0.7925, 0.7952, 0.2159],
        [0.3108, 0.1950, 0.6448, 0.6367],
        [0.0521, 0.2200, 0.5038, 0.9020]])
tensor([[0.3059, 0.2612, 0.9326, 0.5597]])
```

Declare W, x, b

```
hbar = x@W+b
print(hbar)
```

```
tensor([[0.7737, 1.0787, 1.5868, 1.0409]])
```

```
h = torch.nn.Sigmoid()(hbar)
print(h)
```

```
tensor([[0.6843, 0.7463, 0.8302, 0.7390]])
```

Forward propagation

```
v = h*(1-h)
v = v.squeeze()
print(v)
```

```
tensor([0.2160, 0.1894, 0.1410, 0.1929])
```

```
A = torch.diag(v)
print(A)
```

```
tensor([[0.2160, 0.0000, 0.0000, 0.0000],
        [0.0000, 0.1894, 0.0000, 0.0000],
        [0.0000, 0.0000, 0.1410, 0.0000],
        [0.0000, 0.0000, 0.0000, 0.1929]])
```

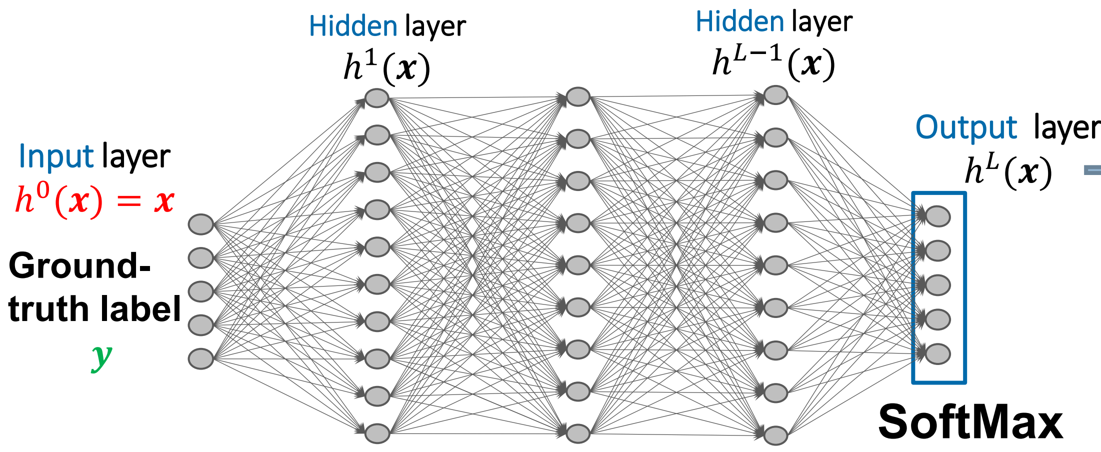
```
derivative = A@W.T
print(derivative)
```

```
tensor([[0.1570, 0.0671, 0.0113],
        [0.1501, 0.0369, 0.0417],
        [0.1121, 0.0909, 0.0710],
        [0.0416, 0.1228, 0.1740]])
```

Backward propagation

Loss Landscape of DL models

Recall optimization problem in deep learning



$\Longrightarrow p(x) = softmax(h^L(x))$

$\hat{y} = \mathrm{argmax}_{1 \le m \le M} p_m(x)$

$l(y, \hat{y}) = CE\left(1_y, p(x)\right) = -\log p_y(x)$

Training set $\longrightarrow$ **Loss function**

$D = \{(x_1, y_1), \ldots (x_N, y_N)\}$

$$L(D; \theta) := \frac{1}{N} \sum_{i=1}^{N} CE\left(1_{y_i}, p(x_i)\right) = -\frac{1}{N} \sum_{i=1}^{N} \log p_{y_i}(x_i)$$

- How to **solve** the **optimization problem** efficiently ($\theta := \left\{\left(W^l, b^l\right)\right\}_{l=1}^{L}$)?
  - $\min_{\theta} L(D; \theta) := -\frac{1}{N} \sum_{i=1}^{N} \log p_{y_i}(x_i) = -\frac{1}{N} \sum_{i=1}^{N} \log \frac{\exp\{h_{y_i}^L(x_i)\}}{\sum_{m=1}^{M} \exp\{h_m^L(x_i)\}}$
  - Generalize: $\min_{\theta} J(\theta) := \frac{1}{N} \sum_{i=1}^{N} l(f(x_i; \theta), y_i)$

Optimization problem in ML and DL

- Most of optimization problems (OP) in machine learning (deep learning) has the following form:

$$\min_\theta J(\theta) = \underbrace{\Omega(\theta)} + \underbrace{\frac{1}{N}\sum_{i=1}^{N} l(y_i, f(x_i;\theta))}$$

Regularization term

- $\Omega(\theta) = \lambda \sum_k \sum_{i,j} \left(W_{i,j}^{\mathbf{k}}\right)^2 = \lambda \sum_k \left\| \boldsymbol{W}^{\mathbf{k}} \right\|_F^2$
- Encourage simple models
- Avoid overfitting

Empirical loss

- Work well on training set

- **Occam's Razor principle**: prefer simplest model that can well predict data.

How to efficiently solve this optimization problem?

N is the **training size** and might be very big (e.g., $N \approx 10^6$)

First-order iterative methods (gradient descent, steepest descent)

Use the **gradient** (first derivative) $\mathrm{g} = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ to update parameters

Second-order iterative methods (Newton and quasi Newton methods)

Use the **Hessian** matrix (second derivative) $\mathrm{H} = \nabla_{\boldsymbol{\theta}}^2 J(\boldsymbol{\theta})$ to update parameters

Gradient and Hessian matrix

- Given an **objective function** $J(\theta)$ with $\theta = [\theta_1, \theta_2, \dots, \theta_P]$
  - For DL models
    - θ includes **weight matrices**, **filters**, and **biases** which are trainable model parameters.
    - P is the number of **trainable parameters** (P could be 20×10^6).
    - $J(\theta)$ is the loss function over a training set.

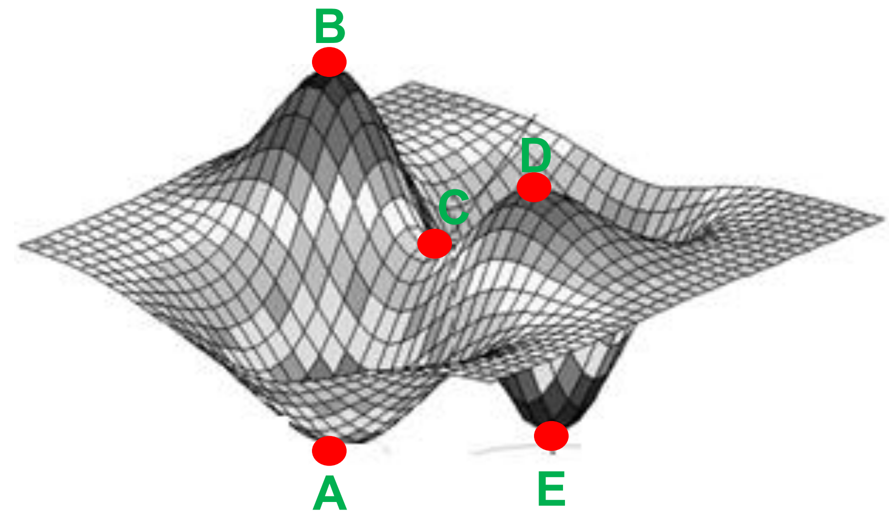
- Gradient $\mathrm{g} = \nabla J(\theta)$ is the **first order derivative** and defined as
  - $\nabla J(\theta) = g = \begin{bmatrix} \frac{\partial J}{\partial \theta_1}(\theta) \\ \dots \dots \dots . \\ \frac{\partial J}{\partial \theta_P}(\theta) \end{bmatrix}$

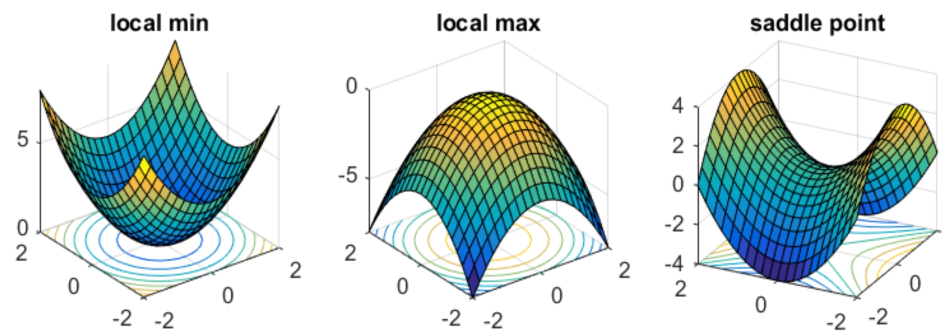
- Hessian matrix $H(\theta)$ is the **second order derivative** $\nabla^2 J(\theta)$ and defined as
  - $$\nabla^2 J(\theta) = H(\theta) = \begin{bmatrix} \frac{\partial^2 J}{\partial \theta_1 \partial \theta_1}(\theta) & \dots & \dots & \frac{\partial^2 J}{\partial \theta_1 \partial \theta_j}(\theta) & \dots & \frac{\partial^2 J}{\partial \theta_1 \partial \theta_P}(\theta) \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \frac{\partial^2 J}{\partial \theta_i \partial \theta_1}(\theta) & \dots & \dots & \frac{\partial^2 J}{\partial \theta_i \partial \theta_j}(\theta) & \dots & \frac{\partial^2 J}{\partial \theta_i \partial \theta_P}(\theta) \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \frac{\partial^2 J}{\partial \theta_P \partial \theta_1}(\theta) & \dots & \dots & \frac{\partial^2 J}{\partial \theta_P \partial \theta_j}(\theta) & \dots & \frac{\partial^2 J}{\partial \theta_P \partial \theta_P}(\theta) \end{bmatrix}$$

MONASH University

Local minima-maxima and saddle point

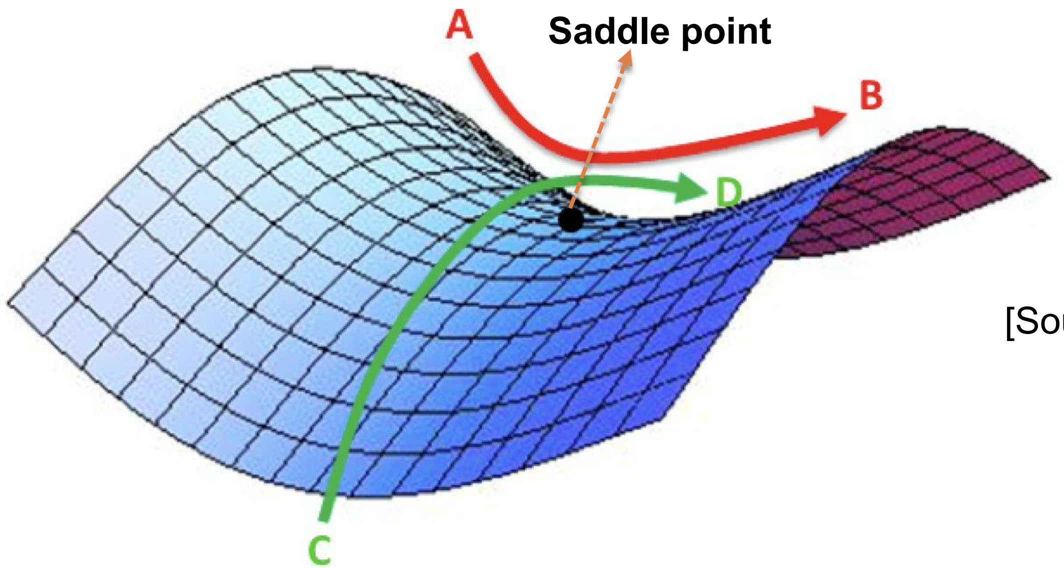
- Given an **objective function** $J(\theta)$ with $\theta = [\theta_1, \theta_2, \dots, \theta_P]$
  - θ is said to be a **critical point** if $\nabla J(\theta) = \mathbf{0}$ (vector $\mathbf{0}$)
- Let us denote the **set of eigenvalues** of Hessian matrix $\nabla^2 J(\theta) = H(\theta)$ by
  - $\lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_P$
- **Local minima**
  - $\nabla J(\theta) = \mathbf{0}$ and $\nabla^2 J(\theta) = H(\theta) \succ 0$ (positive semi-definite matrix)
  - $0 \leq \lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_P$
- **Local maxima**
  - $\nabla J(\theta) = \mathbf{0}$ and $\nabla^2 J(\theta) = H(\theta) \prec 0$ (negative semi-definite matrix)
  - $\lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_P \leq 0$
- **Saddle point**
  - $\nabla J(\theta) = \mathbf{0}$ and $\nabla^2 J(\theta) = H(\theta) \prec 0$ (indefinite matrix)
  - $\lambda_1 \leq \lambda_2 \leq \cdots < 0 < \cdots \leq \lambda_P$





MONASH University

More on saddle point



[Source: Internet]

$$f(\theta) = f(\theta_1, \theta_2) = \theta_1^2 - \theta_2^2$$

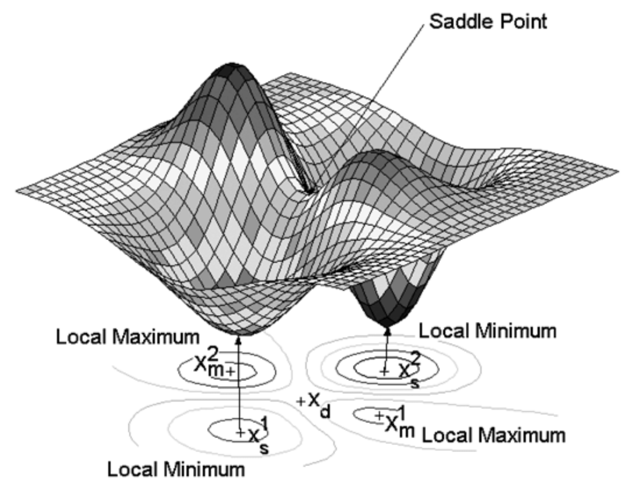
Gradient $g = \begin{bmatrix} \frac{\partial f}{\partial \theta_1} \\ \frac{\partial f}{\partial \theta_2} \end{bmatrix} = \begin{bmatrix} 2\theta_1 \\ -2\theta_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$ → a **critical point** $\begin{bmatrix} 0 \\ 0 \end{bmatrix}$.

Hessian matrix is $H = \begin{bmatrix} \frac{\partial^2 f}{\partial \theta_1^2} & \frac{\partial^2 f}{\partial \theta_1 \partial \theta_2} \\ \frac{\partial^2 f}{\partial \theta_2 \partial \theta_1} & \frac{\partial^2 f}{\partial \theta_2^2} \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 0 & -2 \end{bmatrix}$

Two eigenvalues $\lambda_1 = -2 < 0 < 2 = \lambda_2$ → $\begin{bmatrix} 0 \\ 0 \end{bmatrix}$ is a **saddle point**.

Numbers of local minima vs saddle points

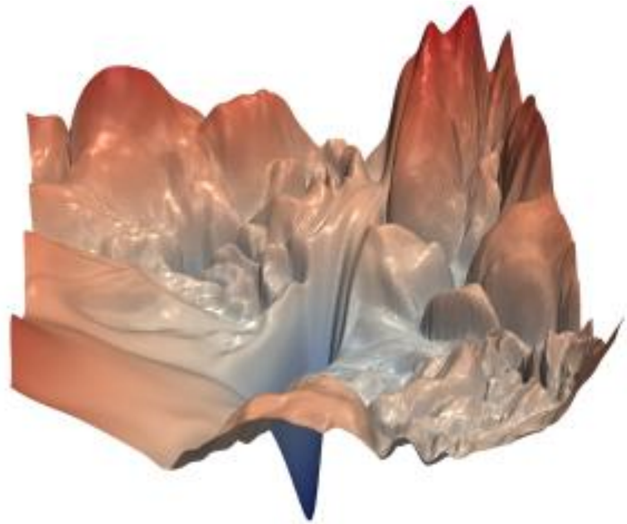
- We assume to pick randomly a training set
  - The Hessian matrix $H(\theta)$ is a random matrix with **random eigenvalues** $\lambda_1, \lambda_2, \dots, \lambda_P$
  - We assume that $\mathbb{P}(\lambda_1 \geq 0) = \mathbb{P}(\lambda_2 \geq 0) = \cdots = \mathbb{P}(\lambda_P \geq 0) = 0.5$



- Therefore, we have
  - $\mathbb{P}(minima) = \mathbb{P}(\lambda_1 \geq 0)\mathbb{P}(\lambda_2 \geq 0) \dots \mathbb{P}(\lambda_P \geq 0) = 0.5^P$
  - $\mathbb{P}(maxima) = \mathbb{P}(\lambda_1 \leq 0)\mathbb{P}(\lambda_2 \leq 0) \dots \mathbb{P}(\lambda_P \leq 0) = 0.5^P$
  - $\mathbb{P}(saddle\ point) = 1 - \mathbb{P}(minima) - \mathbb{P}(maxima) = 1 - 0.5^{P-1}$

- The ratio of **#local minima/maxima against #saddle points**
  - **#local-minima:#local-maxima:#saddle-point=**$1:1:(2^P - 2)$
  - Number of **saddle points** is even **exponentially much more** than that of **local minima/maxima**

The loss surface of DL optimization problem

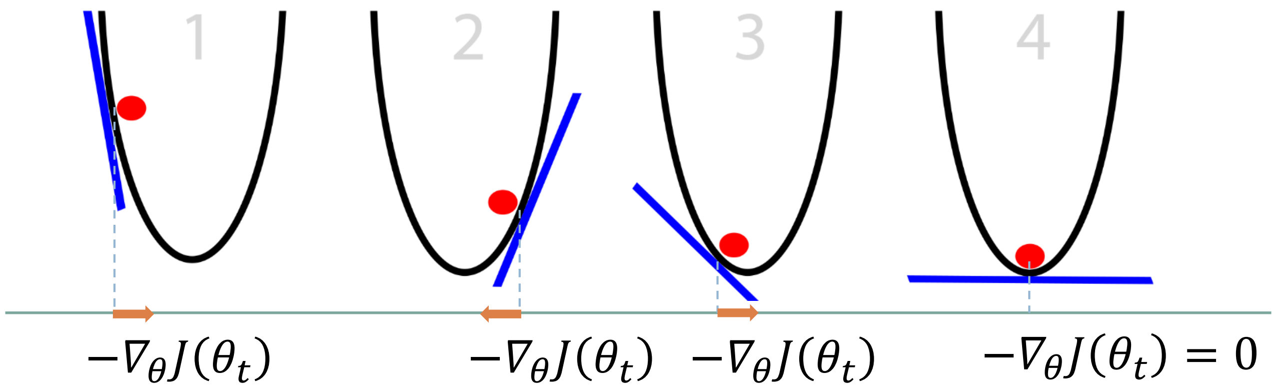


Loss surface of a ResNet without skip connection [Hao Li et al., NeurIPS 2017]

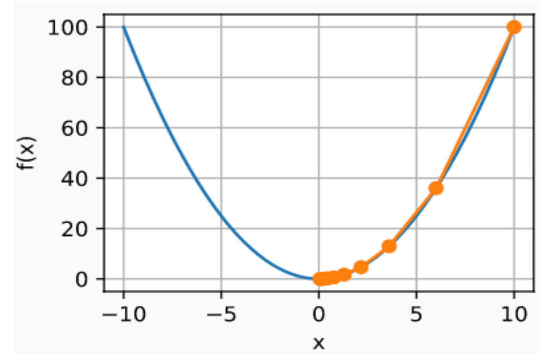
- The **optimization problem** in **deep learning:**
  - $\min_{\theta} J(\theta) \coloneqq L(D;\theta) \coloneqq \frac{1}{N}\sum_{i=1}^{N} l(f(x_i;\theta), y_i) = -\frac{1}{N}\sum_{n=1}^{N} \log \frac{\exp\{h_{y_i}^{L}(x_i)\}}{\sum_{m=1}^{M} \exp\{h_m^{L}(x_i)\}}$
- A very **complex** and **complicated** objective function
  - Highly **non-linear** and **non-convex** function
  - The **loss surface** is very **complex**
  - Many local minima points, but the number of saddle points is even **exponentially much more**

Gradient descent and stochastic gradient descent

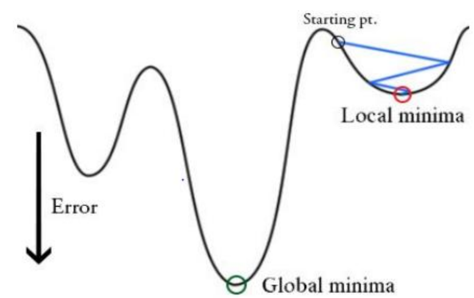
Gradient descend



- We need to solve
  - $\min_{\theta} J(\theta)$
- Follow to **the opposite side** of the current gradient
  - $\theta_{t+1} = \theta_t - \eta \nabla_\theta J(\theta_t)$ where $\boldsymbol{\eta} > \mathbf{0}$ is the **learning rate**.
- Guarantee to converge to a **global minima** if $J(.)$ is **convex**.
- Get stuck in a **local minima** or **saddle points** if $J(.)$ is non-convex.

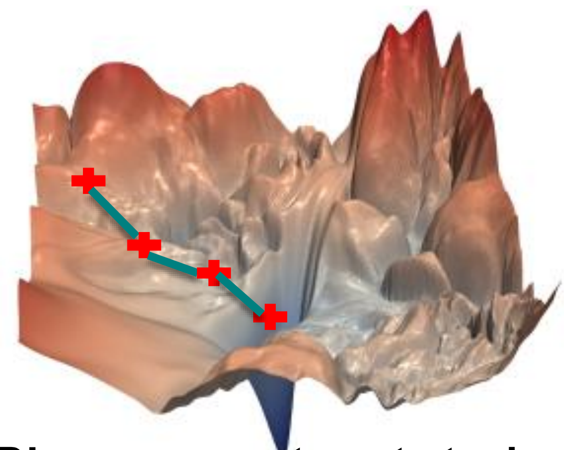


Convex case



(Source: www.cs.ubc.ca)

Non-convex case



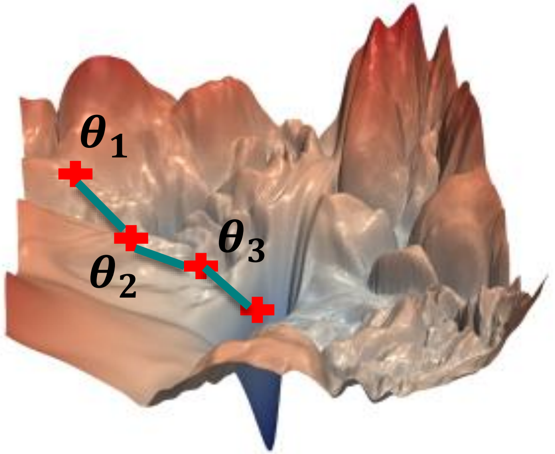
DL case: easy to get stuck in saddle points

Gradient descend

Algorithm

- **Input**: objective function $\boldsymbol{J}(\boldsymbol{\theta})$
- **Output**: optimal solution $\boldsymbol{\theta}^*$

1. Initialize parameters θ_0 randomly $\sim N(0, \sigma^2)$.
2. for t=1 to T
3. Compute gradients $\nabla_\theta J(\theta_t) = \frac{\partial J}{\partial \theta}(\theta_t)$
4. Update $\theta_{t+1} = \theta_t - \eta_t \nabla_\theta J(\theta_t)$
5. Return $\theta^* = \theta_{T+1}$

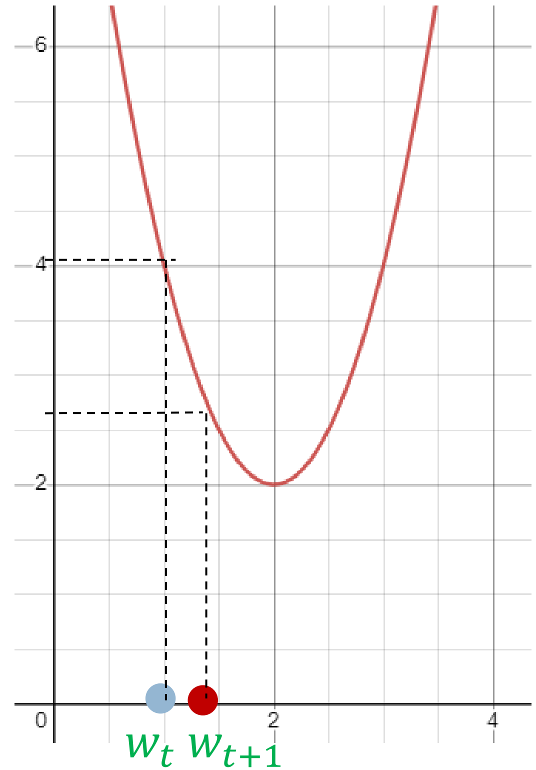


Example of Gradient Descend

- Consider the optimization problem:

$$\min_w [f(w) = (w-1)^2 + (w-3)^2]$$

- Currently, we are at $w_t = 1$ with $f(w_t) = f(1) = 0^2 + 2^2 = 4$. What is w_{t+1} if using learning rate $\eta = 0.1$?
  - $f'(w) = 2(w-1) + 2(w-3) = 4w - 8$
  - $f'(w_t) = f'(1) = -4$
  - $w_{t+1} = w_t - \eta f'(w_t) = 1 - 0.1(-4) = 1.4$
  - $f(w_{t+1}) = 2.72 < f(w_t) = 4$



Gradient descent for deep learning

- For **training deep nets**, we need to solve
  - $\min_{\theta} L(D;\theta) := \frac{1}{N}\sum_{i=1}^{N} l(x_i, y_i; \theta) = \frac{1}{N}\sum_{i=1}^{N} l(y_i, f(x_i;\theta))$

  where $l(x_i, y_i; \theta) = -\log p(y = y_i | x_i) = -\log \frac{\exp\{h_{y_i}^{L}(x_i)\}}{\sum_{m=1}^{M} \exp\{h_m^{L}(x_i)\}}$ is the loss incurred by (x_i, y_i).
- Gradient descent update
  - $\theta_{t+1} = \theta_t - \eta \nabla_\theta L(D;\theta_t) = \theta_t - \frac{\eta}{N}\sum_{i=1}^{N} \nabla_\theta\, l(x_i, y_i; \theta_t)$ where $\boldsymbol{\eta} > \mathbf{0}$ is a learning rate.
  - To compute the gradient $\nabla_\theta L(D;\theta_t) = \frac{1}{N}\sum_{i=1}^{N} \nabla_\theta\, l(x_i, y_i; \theta_t)$, we need to go through **all data points** in D → the **computational cost** is $O(N)$.
- This is very **computationally expensive** for big datasets ($N \approx 10^6$).
- How to **estimate the gradient** $\nabla_\theta L(D;\theta_t)$ more efficiently?

Stochastic gradient descent

- The **optimization problem** in **deep learning** has the form
  - $\min_{\theta} L(D;\theta) := \frac{1}{N}\sum_{i=1}^{N} l(x_i, y_i;\theta) \;\; = \frac{1}{N}\sum_{i=1}^{N} l(y_i, f(x_i;\theta))$
- Evaluation of the **full gradient** is **expensive**. We want to just **estimate** this gradient
  - Sample a mini-batch $\boldsymbol{i_1, i_2, \dots, i_b \sim Uni(\{1,2,\dots,N\})}$ where b is the mini-batch (batch) size.
    - The batch size is usually $32, 64, 128, 256$, and so on.
  - Construct $\tilde{L}(\theta) := \frac{1}{b}\sum_{k=1}^{b} l(x_{i_k}, y_{i_k};\theta)$ as the average loss of those in the current batch.
  - $E_{i_1,\dots,i_b}[\nabla_\theta \tilde{L}(\theta_t)] = \nabla_\theta L(D;\theta_t)$
    - $\nabla_\theta \tilde{L}(\theta_t) = \frac{1}{b}\sum_{k=1}^{b} \nabla_\theta l(x_{i_k}, y_{i_k};\theta_t)$ is **unbiased** estimation of $\nabla_\theta L(D;\theta_t)$
    - $O(b)$ compares to $O(N)$.
- The update rule of SGD
  - $\theta_{t+1} = \theta_t - \eta_t \nabla_\theta \tilde{L}(\theta_t)$ with learning rate $\boldsymbol{\eta_t \propto O(\frac{1}{t})}$
  - We use $\nabla_\theta \tilde{L}(\theta_t)$ as an **unbiased estimate** of the full gradient $\nabla_\theta L(D;\theta)$
  - How to compute $\nabla_\theta \tilde{L}(\theta_t)$ efficiently for **deep networks**?

Example of Stochastic Gradient Descent

- Given the function $f(w) = \frac{1}{1000}\sum_{i=1}^{1000}(w-i)^2$, we need to solve

$$\min_{w} f(w)$$

using SGD with the learning rate $\eta = 0.1$.

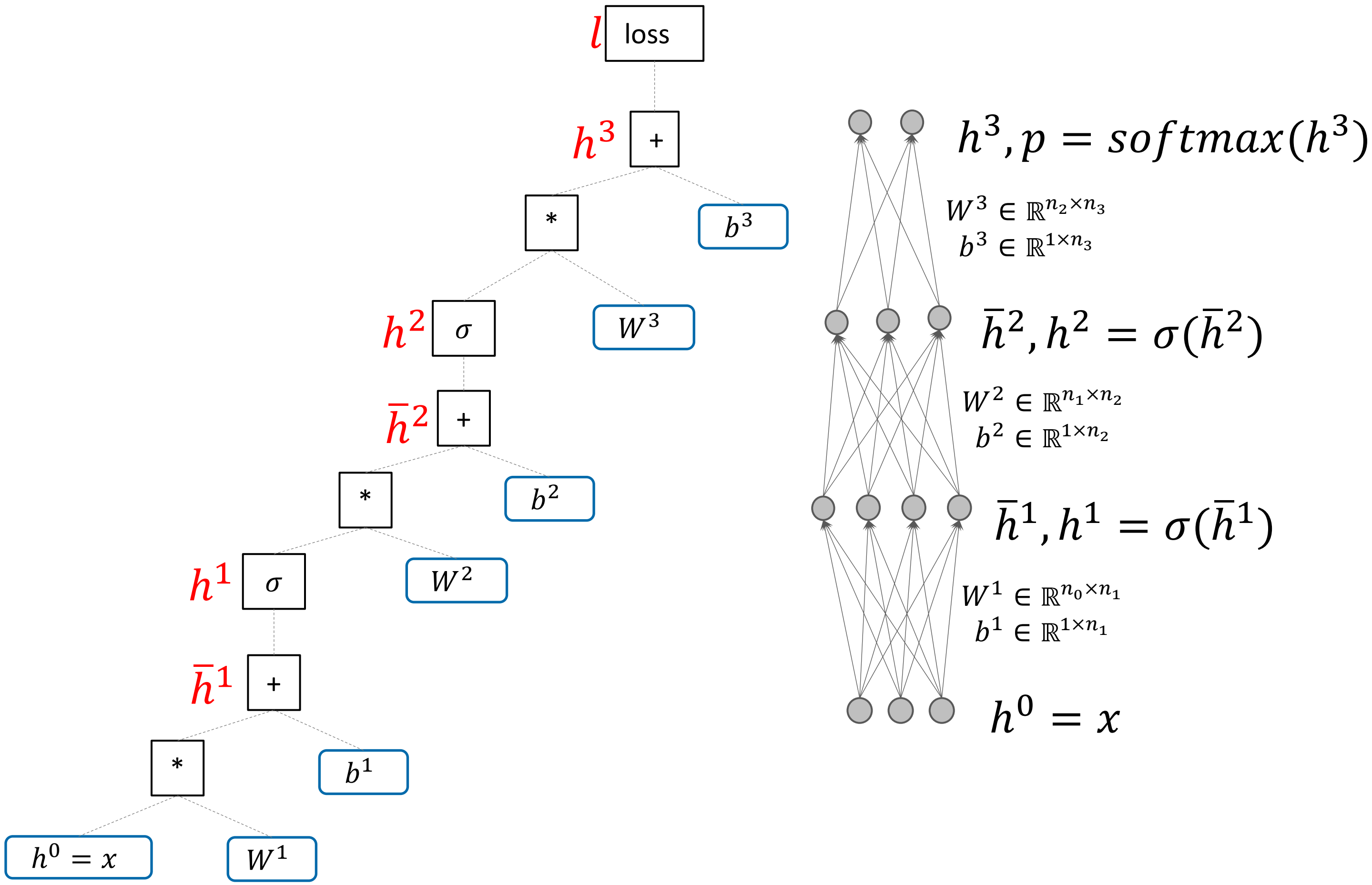
- Assume we sample a batch $i_1 = 1, i_2 = 2, i_3 = 3, i_4 = 4$ of indices. At the iteration t, $w_t = 10$. What is the value of w_{t+1} at the next iteration?

- $\tilde{f}(w) = \frac{1}{4}[(w-1)^2 + (w-2)^2 + (w-3)^4 + (w-4)^2]$
- $\tilde{f}'(w) = 2w - 5$
- $\tilde{f}'(w_t) = \tilde{f}'(10) = 2 \times 10 - 5 = 15$
- $w_{t+1} = w_t - \eta \tilde{f}'(w_t) = 10 - 0.1 \times 15 = 8.5.$


Q&A
When? Why? How? WHERE? WHEN? What? How? Why? WHAT? Why? When?
WHAT? HOW? WHEN? When? WHERE?
When? WHERE? WHAT? Why?
When?
WHO? WHAT?
WHERE? WHEN?
What? HOW?
Why? HOW? Where? Where? WHERE? When? What? What? WHERE? WHERE? When?
Why? What? When?
WHEN? WHEN? HOW? When? When? HOW? WHAT? Why?

Back propagation in feed-forward neural networks

Back propagation

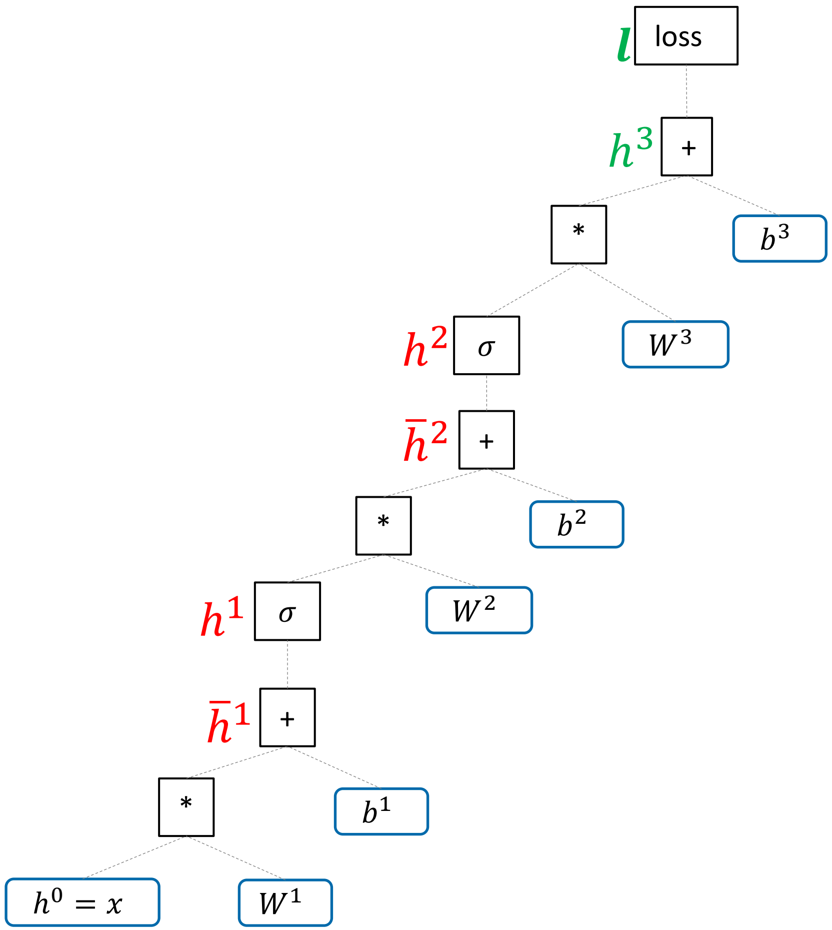


- Given a data point and label pair (x, y)
  - $l(x, y; \theta) = -\log \frac{\exp\{h_y^3(x)\}}{\sum_{m=1}^{M} \exp\{h_m^3(x)\}}$
- What are the derivatives?
  - $\nabla_{W^k} l(x, y; \theta)$ and $\nabla_{b^k} l(x, y; \theta)$ for $k = 1,2,3$?
  - Using **back propagation** to compute **these derivatives** conveniently.
- Update the model using SGD with the derivatives.

Back propagation

From loss to h^3

$$f(x,y,z) = \log(\exp(x) + \exp(y) + \exp(z))$$
$$\nabla f = [f'_x, f'_y, f'_z] = softmax([x,y,z])$$



- $l(x,y;\theta) = -\log \frac{\exp\{h_y^3(x)\}}{\sum_{m=1}^{M} \exp\{h_m^3(x)\}} =$
  $-h_y^3(x) + \log[\sum_{m=1}^{M} \exp\{h_m^3(x)\}] =$
  $-\sum_{m=1}^{M} \mathbf{1}_{m=y} h_m^3 + \log[\sum_{m=1}^{M} \exp\{h_m^3\}]$

where $\mathbf{1}_{m=y} = \mathbf{1}$ if $m = y$ and $\mathbf{0}$ otherwise.

- $\frac{\partial l}{\partial h_m^3} = -\mathbf{1}_{m=y} + \frac{exp\{h_m^3\}}{\sum_{k=1}^{M} exp\{h_k^3\}}, m = 1,..,M$

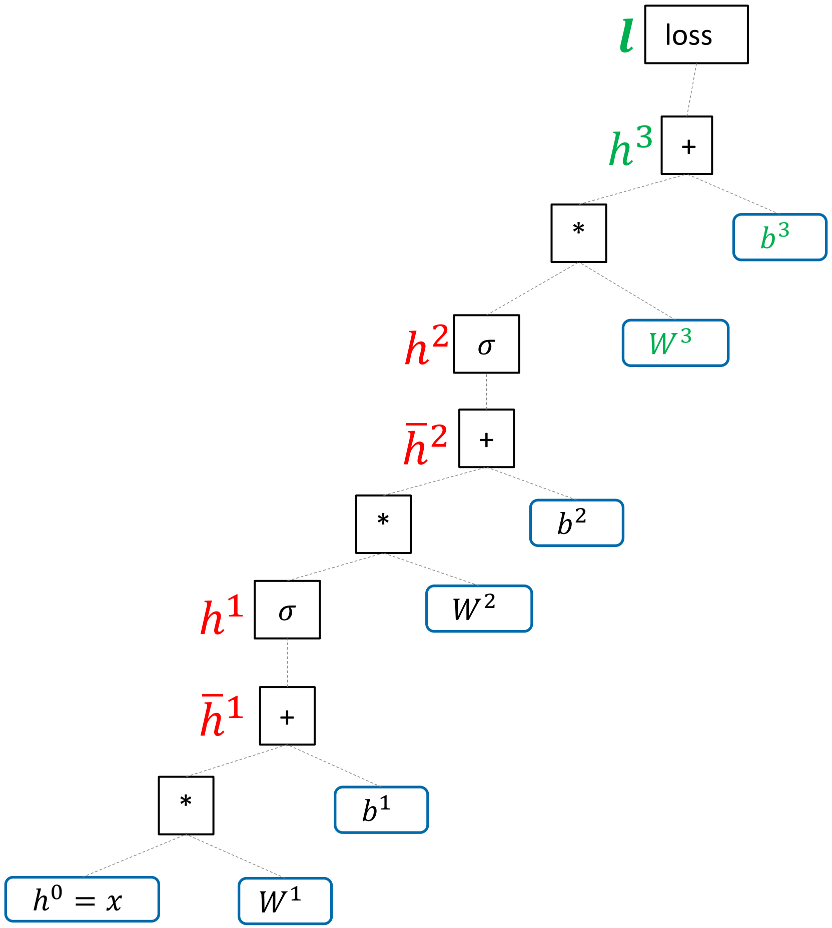
- $g^3 = \frac{\partial l}{\partial h^3} = -\mathbf{1}_y + softmax(h^3) = p - \mathbf{1}_y$

where $\mathbf{1}_y$ is the corresponding one-hot vector.

  - g^3 has a shape $[1 \times n_3]$.

Back propagation

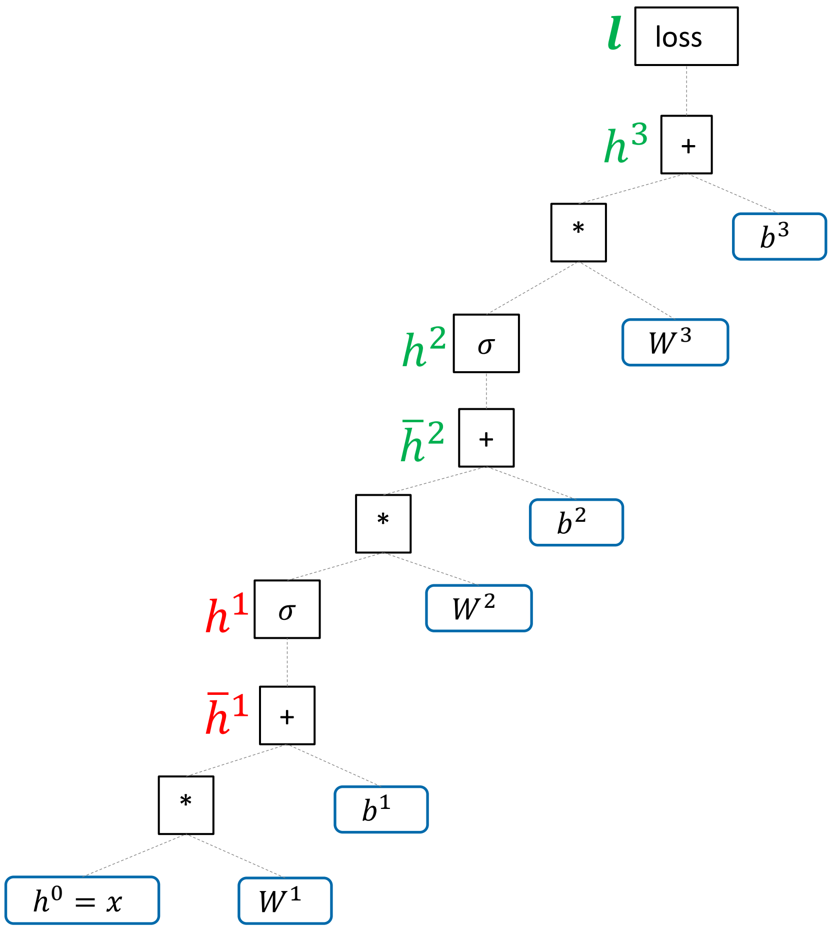
From loss to W^3, b^3



- $h^3 = h^2 W^3 + b^3$
- $\frac{\partial l}{\partial W^3} = \frac{\partial l}{\partial h^3} . \frac{\partial h^3}{\partial W^3} = \left(h^2\right)^T g^3$
  - $[n_2 \times 1] \times [1 \times n_3] \rightarrow [n_2 \times n_3]$
- $\frac{\partial l}{\partial b^3} = \frac{\partial l}{\partial h^3} . \frac{\partial h^3}{\partial b^3} = g^3$
  - $[1 \times n_3]$

Back propagation

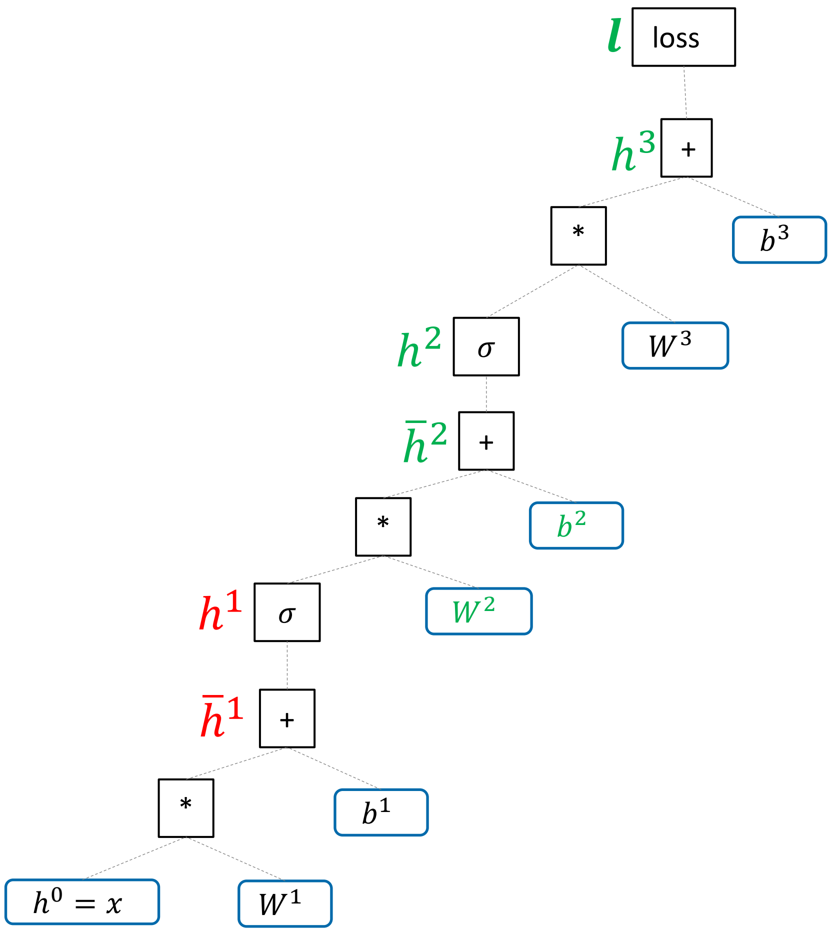
From loss to h^2 and $\bar{h}^2$



- $h^3 = h^2 W^3 + b^3$
- $\boldsymbol{g}^2 = \frac{\partial \boldsymbol{l}}{\partial \boldsymbol{h}^2} = \frac{\partial \boldsymbol{l}}{\partial \boldsymbol{h}^3} \cdot \frac{\partial \boldsymbol{h}^3}{\partial \boldsymbol{h}^2} = \boldsymbol{g}^3 (\boldsymbol{W}^3)^{\boldsymbol{T}}$
  - $[1 \times n_3] \times [n_3 \times n_2] \to [1 \times n_2]$
- $\boldsymbol{h}^2 = \boldsymbol{\sigma}(\bar{\boldsymbol{h}}^2)$ (element-wise activation)
- $\frac{\partial \boldsymbol{h}^2}{\partial \bar{\boldsymbol{h}}^2} = \boldsymbol{diag}(\boldsymbol{\sigma}'(\bar{\boldsymbol{h}}^2))$
  - $\sigma'(\bar{h}^2)$ is **element-wise derivative** and $diag(u)$ is the **diagonal matrix** corresponding to the **vector $\boldsymbol{u}$** (the diagnose is u and others are zeros).
- $\bar{\boldsymbol{g}}^2 = \frac{\partial \boldsymbol{l}}{\partial \bar{\boldsymbol{h}}^2} = \frac{\partial \boldsymbol{l}}{\partial \boldsymbol{h}^2} \cdot \frac{\partial \boldsymbol{h}^2}{\partial \bar{\boldsymbol{h}}^2} = \boldsymbol{g}^2 \boldsymbol{diag}(\boldsymbol{\sigma}'(\bar{\boldsymbol{h}}^2))$
  - $[1 \times n_2] \times [n_2 \times n_2] \to [1 \times n_2]$

Back propagation

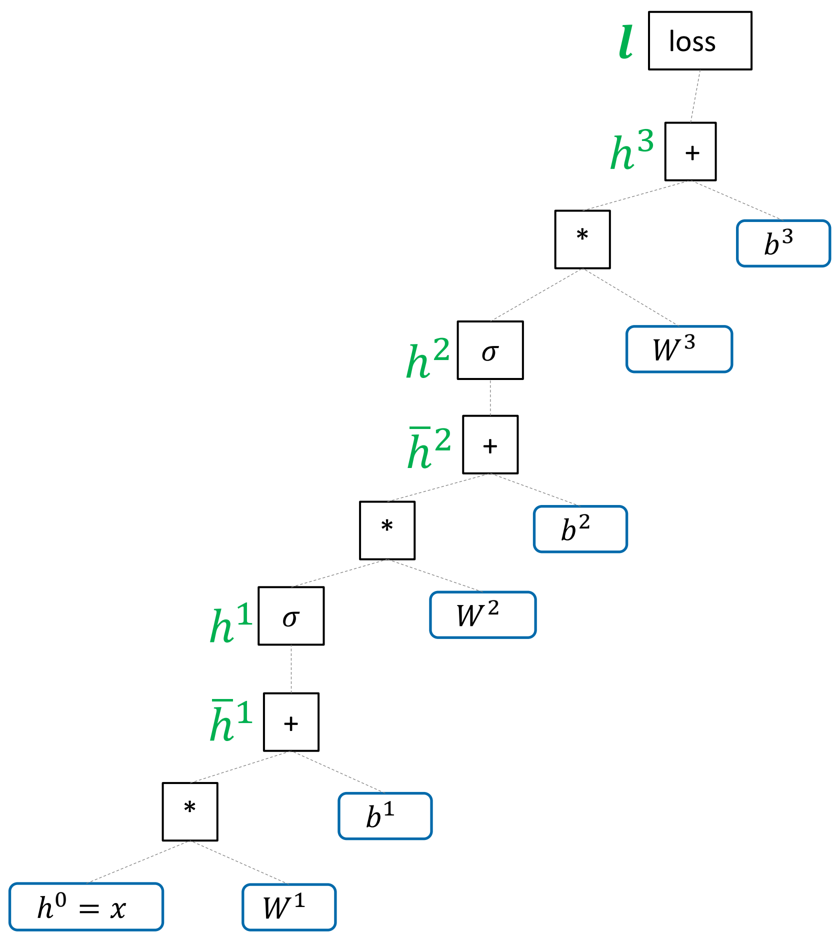
From loss to W^2 and b^2



- $\bar{h}^2 = h^1 W^2 + b^2$
- $\dfrac{\partial l}{\partial W^2} = \dfrac{\partial l}{\partial \bar{h}^2} \cdot \dfrac{\partial \bar{h}^2}{\partial W^2} = (h^1)^T \bar{g}^2$
  - $[n_1 \times 1] \times [1 \times n_2] \to [n_1 \times n_2]$
- $\dfrac{\partial l}{\partial b^2} = \dfrac{\partial l}{\partial \bar{h}^2} \cdot \dfrac{\partial \bar{h}^2}{\partial b^2} = \bar{g}^2$
  - $[1 \times n_2]$

Back propagation

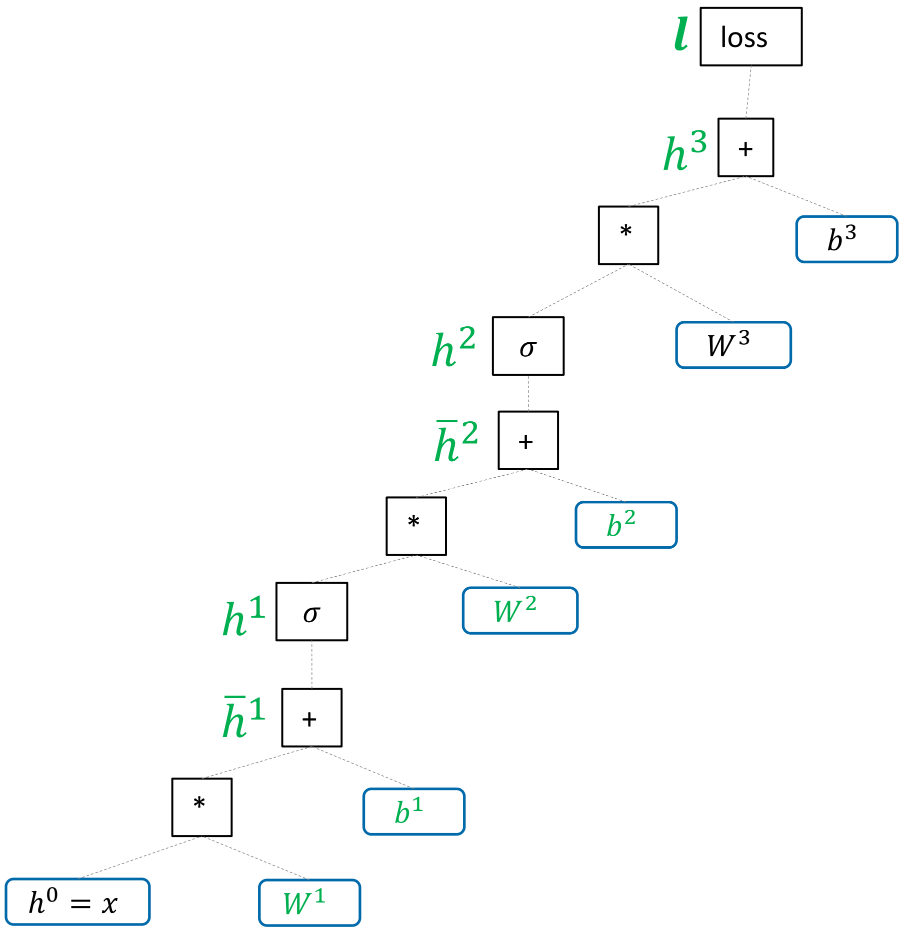
From loss to h^1 and $\bar{h}^1$



- $\bar{h}^2 = h^1 W^2 + b^2$
- $\boldsymbol{g^1 = \frac{\partial l}{\partial h^1} = \frac{\partial l}{\partial \bar{h}^2} \cdot \frac{\partial \bar{h}^2}{\partial h^1} = \bar{g}^2 (W^2)^T}$
  - $[1 \times n_2] \times [n_2 \times n_1] \to [1 \times n_1]$
- $\boldsymbol{h^1 = \sigma(\bar{h}^1)}$ (element-wise activation)
- $\boldsymbol{\frac{\partial h^1}{\partial \bar{h}^1} = diag(\sigma'(\bar{h}^1))}$
  - $\sigma'(\bar{h}^1)$ is **element-wise derivative** and $diag(u)$ is the **diagonal matr**ix corresponding to the **vector** $\boldsymbol{u}$ (the diagnose is u and others are zeros).
- $\boldsymbol{\bar{g}^1 = \frac{\partial l}{\partial \bar{h}^1} = \frac{\partial l}{\partial h^1} \cdot \frac{\partial h^1}{\partial \bar{h}^1} = g^1 diag(\sigma'(\bar{h}^1))}$
  - $[1 \times n_1] \times [n_1 \times n_1] \to [1 \times n_1]$

Back propagation

From loss to W^1 and b^1



- $\bar{h}^1 = h^0 W^1 + b^1 \quad (h^0 = x)$
- $\dfrac{\partial l}{\partial W^1} = \dfrac{\partial l}{\partial \bar{h}^1} \cdot \dfrac{\partial \bar{h}^1}{\partial W^1} = \left(h^0\right)^T \bar{g}^1$
  - $[n_0 \times 1] \times [1 \times n_1] \to [d = n_0 \times n_1]$
- $\dfrac{\partial l}{\partial b^1} = \dfrac{\partial l}{\partial \bar{h}^1} \cdot \dfrac{\partial \bar{h}^1}{\partial b^1} = \bar{g}^1$
  - $[1 \times n_1]$

Exercise: How to compute $\frac{\partial l}{\partial x}$?

SGD for deep learning

b = 32 //batch size

iter_per_epoch = N/b //epoch means one round going through all data points

n_epoch = 50 //number of epochs

for epoch=1 to n_epoch do

for i=1 to iter_per_epoch do

Sample a minibatch $B = \left\{\left(x_{i_j}, y_{i_j}\right)\right\}_{j=1}^{b}$ from the training set

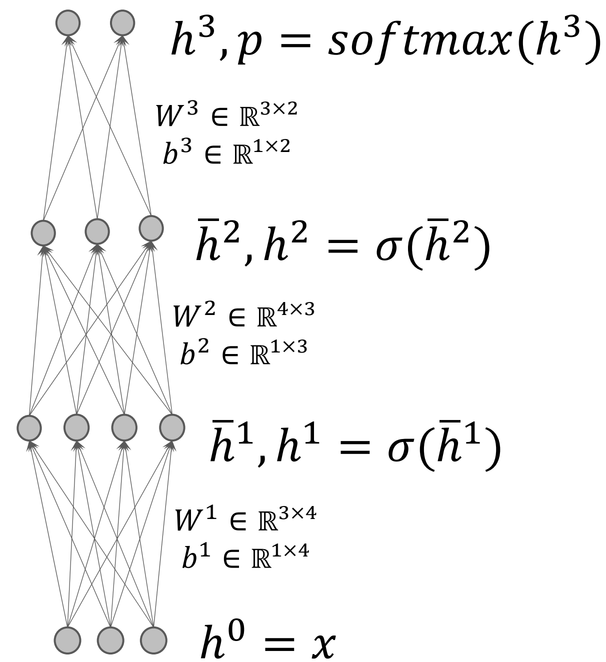
Do forward propagation for B

Do back propagation to compute $\left(\frac{\partial l}{\partial W^k}, \frac{\partial l}{\partial b^k}\right)_{k=1}^{L}$

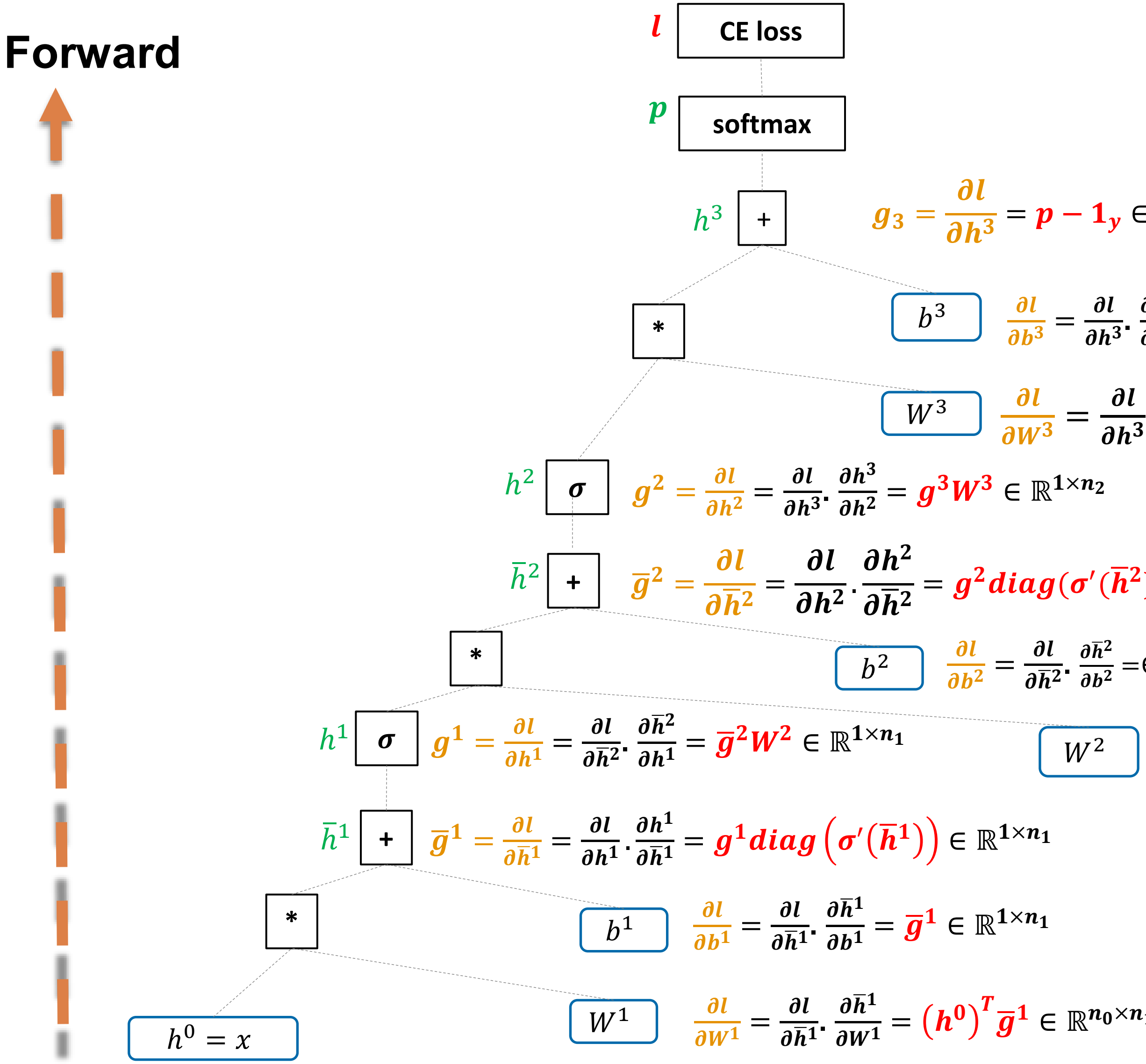
for k=1 to L do

$$W_k = W_k - \eta \frac{\partial l}{\partial W^k}$$

$$b_k = b_k - \eta \frac{\partial l}{\partial b^k}$$



Forward – Backward Propagations



$$g_3 = \frac{\partial l}{\partial h^3} = p - 1_y \in \mathbb{R}^{1\times n_3}$$

$$\frac{\partial l}{\partial b^3} = \frac{\partial l}{\partial h^3}.\frac{\partial h^3}{\partial b^3} = g^3 \in \mathbb{R}^{1\times n_3} \Longrightarrow b^3 = b^3 - \eta\frac{\partial l}{\partial b^3}$$

$$\frac{\partial l}{\partial W^3} = \frac{\partial l}{\partial h^3}.\frac{\partial h^3}{\partial W^3} = (h^2)^T g^3 \in \mathbb{R}^{n_2\times n_3} \Longrightarrow W^3 = W^3 - \eta\frac{\partial l}{\partial W^3}$$

$$g^2 = \frac{\partial l}{\partial h^2} = \frac{\partial l}{\partial h^3}.\frac{\partial h^3}{\partial h^2} = g^3 W^3 \in \mathbb{R}^{1\times n_2}$$

$$\bar{g}^2 = \frac{\partial l}{\partial \bar{h}^2} = \frac{\partial l}{\partial h^2}.\frac{\partial h^2}{\partial \bar{h}^2} = g^2 diag(\sigma'(\bar{h}^2)) \in \mathbb{R}^{1\times n_2}$$

$$\frac{\partial l}{\partial b^2} = \frac{\partial l}{\partial \bar{h}^2}.\frac{\partial \bar{h}^2}{\partial b^2} = \in \mathbb{R}^{1\times n_2} \Longrightarrow b^2 = b^2 - \eta\frac{\partial l}{\partial b^2}$$

$$g^1 = \frac{\partial l}{\partial h^1} = \frac{\partial l}{\partial \bar{h}^2}.\frac{\partial \bar{h}^2}{\partial h^1} = \bar{g}^2 W^2 \in \mathbb{R}^{1\times n_1}$$

$$\frac{\partial l}{\partial W^2} = \frac{\partial l}{\partial \bar{h}^2}.\frac{\partial \bar{h}^2}{\partial W^2} = (h^1)^T \bar{g}^2 \in \mathbb{R}^{n_1\times n_2} \Longrightarrow W^2 = W^2 - \eta\frac{\partial l}{\partial W^2}$$

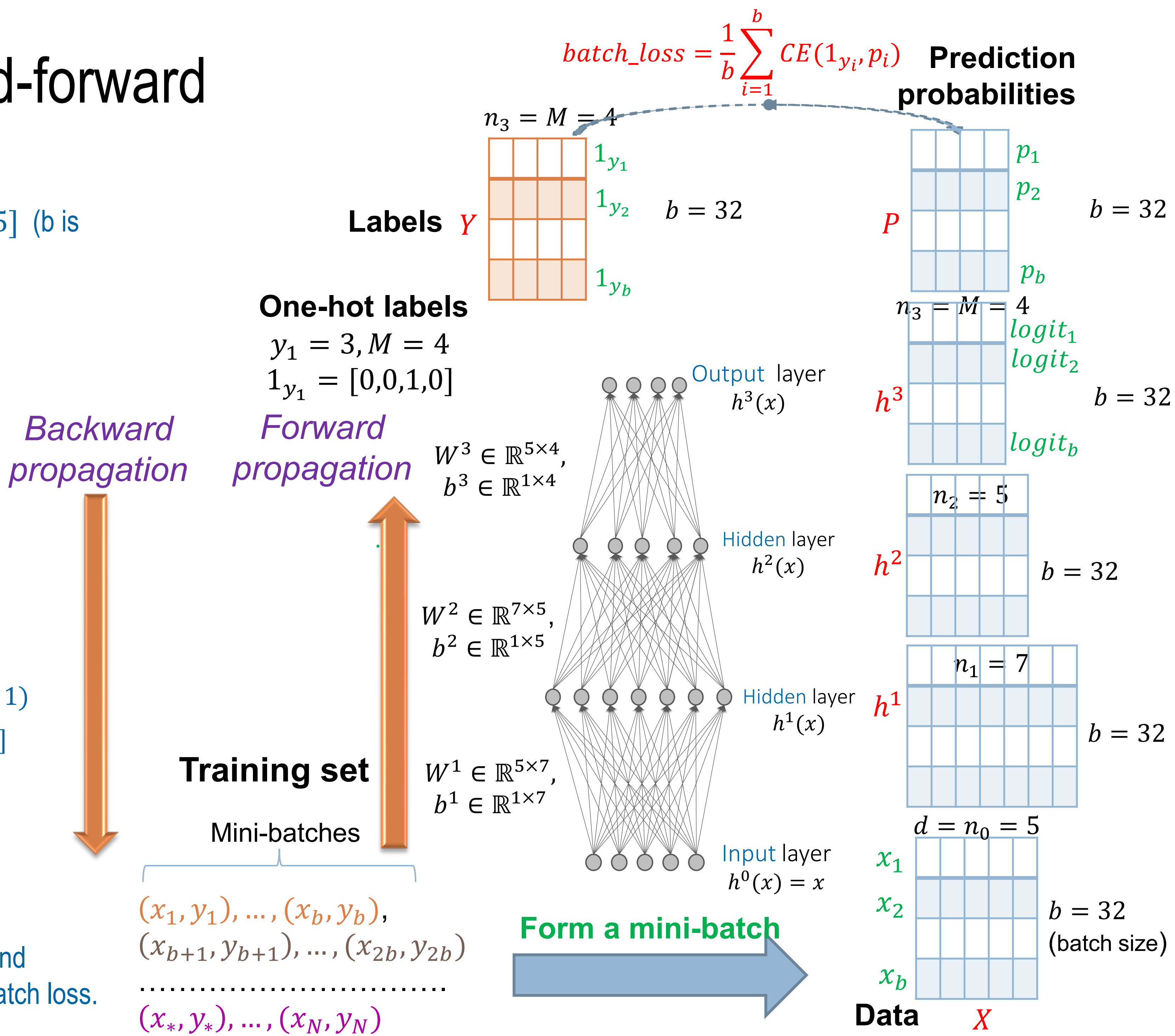
$$\bar{g}^1 = \frac{\partial l}{\partial \bar{h}^1} = \frac{\partial l}{\partial h^1}.\frac{\partial h^1}{\partial \bar{h}^1} = g^1 diag\left(\sigma'(\bar{h}^1)\right) \in \mathbb{R}^{1\times n_1}$$

$$\frac{\partial l}{\partial b^1} = \frac{\partial l}{\partial \bar{h}^1}.\frac{\partial \bar{h}^1}{\partial b^1} = \bar{g}^1 \in \mathbb{R}^{1\times n_1} \Longrightarrow b^1 = b^1 - \eta\frac{\partial l}{\partial b^1}$$

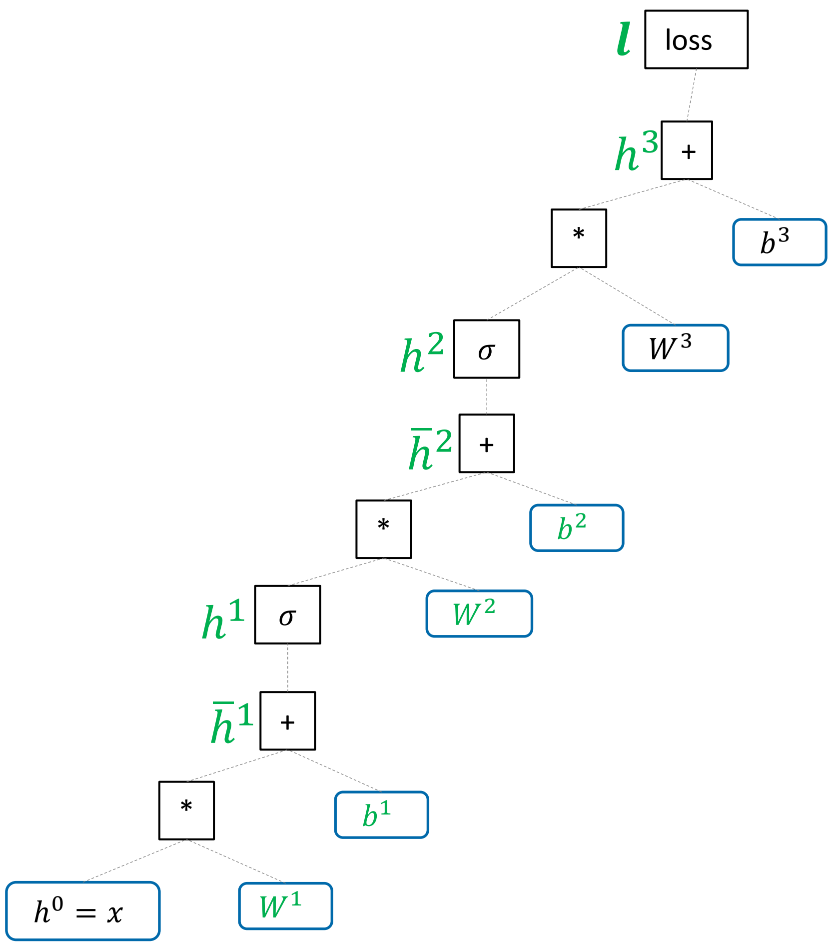
$$\frac{\partial l}{\partial W^1} = \frac{\partial l}{\partial \bar{h}^1}.\frac{\partial \bar{h}^1}{\partial W^1} = (h^0)^T \bar{g}^1 \in \mathbb{R}^{n_0\times n_1} \Longrightarrow W^1 = W^1 - \eta\frac{\partial l}{\partial W^1}$$

Mini-batch feed-forward

- **Input**
  - Batch X: $[b, n_0 = d = 5]$ (b is the batch size)
- **Hidden layer 1**
  - $\boldsymbol{h^1 = \sigma(XW^1 + b^1)}$
  - $Tensor\,[b, n_1 = 7]$
- **Hidden layer 2**
  - $\boldsymbol{h^2 = \sigma(h^1W^2 + b^2)}$
  - Tensor $[b, n_2 = 5]$
- **Output layer**
  - $\boldsymbol{h^3 = h^2W^3 + b^3}$
  - $P = softmax(h^3, dim = 1)$
  - $Tensor$ P: $[b, n_L = M = 4]$
- **The loss of the batch**
  - $\frac{1}{b}\sum_{i=1}^{b} CE(1_{y_i}, p_i) = -\frac{1}{b}\sum_{i=1}^{b} \log p_{y_i}^{i}$
  - Update weight matrices and biases to **minimize** the batch loss.



Why does deep learning need GPU and TPU?



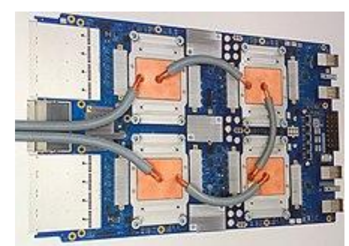
- Let consider

$$\frac{\partial l}{\partial W^1} = \frac{\partial l}{\partial h^3} \cdot \frac{\partial h^3}{\partial h^2} \cdot \frac{\partial h^2}{\partial \bar{h}^2} \cdot \frac{\partial \bar{h}^2}{\partial h^1} \cdot \frac{\partial h^1}{\partial \bar{h}^1} \cdot \frac{\partial \bar{h}^1}{\partial W^1}$$

$$= \left(h^0\right)^T \left(p - 1_y\right) \left(W^3\right)^T diag\left(\sigma'\left(\bar{h}^2\right)\right) \left(W^2\right)^T diag\left(\sigma'\left(\bar{h}^1\right)\right)$$

- For a deep net, this **back propagation** requires many **matrix multiplications**
  - We need specific hardware that can parallel and significantly speed up matrix multiplication operation
  - **GPU** (Graphic Processing Unit) and **TPU** (Tensor Processing Unit)

GPU (Source: HelloTech)

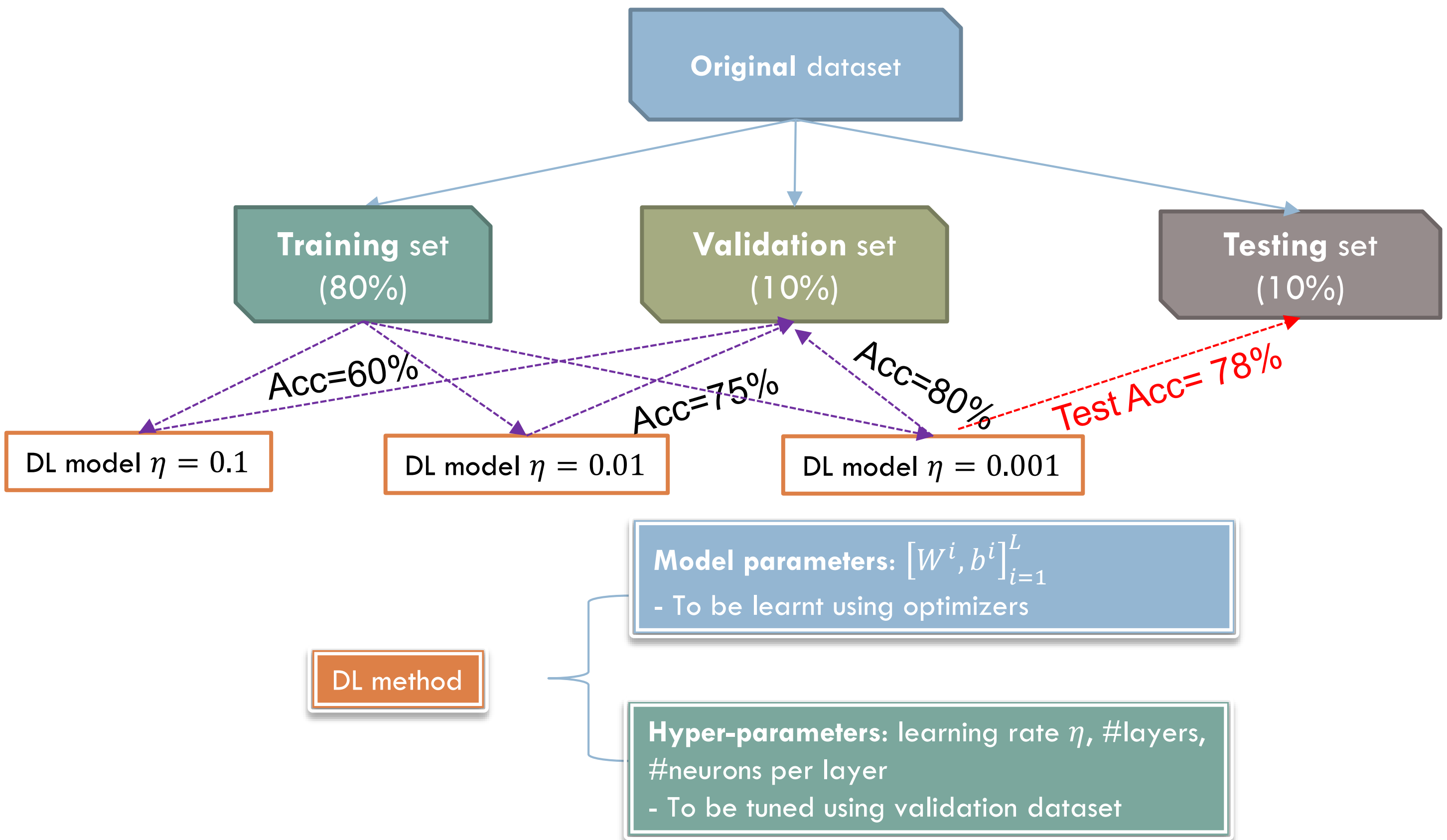


TPU (Source: Wikepedia)

Deep learning pipeline

Tuning hyper-parameters

- We want to train our DL model on a **training set** such that the **trained model** can predict well **unseen data** in a **separate testing set**.

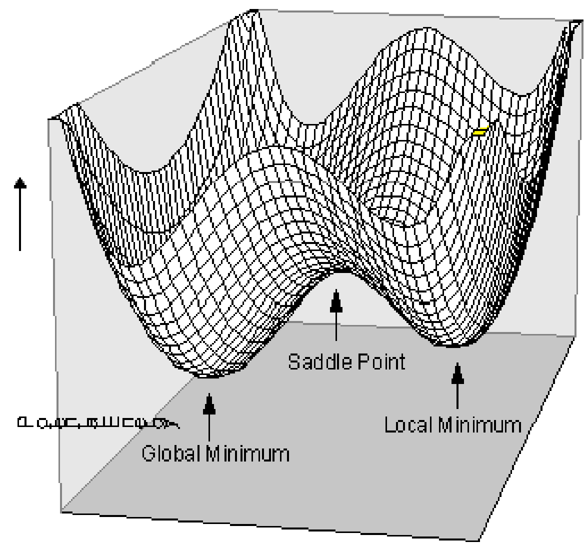



Q&A
When?
Why?
How?
WHAT?
WHERE?
WHEN?
What?
HOW?
Why?
WHAT?
Why?
When?
HOW?
When?
WHERE?
When?
WHEN?
Why?
When?
WHAT?
WHERE?
WHO?
WHAT?
WHEN?
WHERE?
What?
Why?
HOW?
Where?
Where?
WHERE?
WHEN?
HOW?
When?
What?
What?
WHERE?
WHERE?
When?
When?
HOW?
When?
When?
HOW?
WHAT?
Why?
Why?
WHEN?

Optimizers for deep learning

Challenges of optimization for Deep Learning

- The **optimization problem** in **deep learning:**
  - $\min_{\theta} J(\theta) := L(D;\theta) := \frac{1}{N}\sum_{i=1}^{N} l(x_i, y_i; \theta) = -\frac{1}{N}\sum_{n=1}^{N} \log \frac{\exp\{h_{y_i}^{L}(x_i)\}}{\sum_{m=1}^{M} \exp\{h_m^{L}(x_i)\}}$
- A very **complex** and **complicated** objective function
  - Highly **non-linear** and **non-convex** function
  - The **loss surface** is very **complex**
  - Many local minima points, but the number of saddle points is even **exponentially much more**
- Need **efficient optimizers** to solve
  - SGD with momentum, Adagrad, Adadelta, RMSProp, Adam, and Nadam
  - They are **built-in optimizers** of PyTorch.



(Source: Jan Jakubik)

SGD and SGD with momentum

SGD

- **Input**: $\eta > 0$ and *initial model* θ

while ***stopping criterion*** **not met do**

  Sample a mini-batch $\{(\boldsymbol{x}^1, \boldsymbol{y}^1), \dots, (\boldsymbol{x}^b, \boldsymbol{y}^b)\}$

  Compute $\boldsymbol{g} = \frac{1}{b}\sum_{i=1}^{b} \nabla_{\theta} l(f(\boldsymbol{x}^i, \boldsymbol{\theta}), \boldsymbol{y}^i)$

  Update $\boldsymbol{\theta} = \boldsymbol{\theta} - \eta \boldsymbol{g}$

end while

SGD with momentum

- **Input**: $\eta > 0, \alpha \in [0,1]$ and *initial model* θ

while ***stopping criterion*** **not met do**

  Sample a mini-batch $\{(\boldsymbol{x}^1, \boldsymbol{y}^1), \dots, (\boldsymbol{x}^b, \boldsymbol{y}^b)\}$

  Compute $\boldsymbol{g} = \frac{1}{b}\sum_{i=1}^{b} \nabla_{\theta} l(f(\boldsymbol{x}^i, \boldsymbol{\theta}), \boldsymbol{y}^i)$

  Compute $\boldsymbol{v} = \alpha \boldsymbol{g} + (1 - \alpha)\boldsymbol{v}$ //velocity v

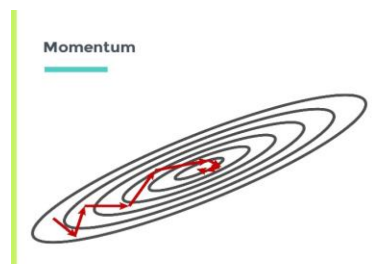
  Update $\boldsymbol{\theta} = \boldsymbol{\theta} - \eta \boldsymbol{v}$

end while

- SGD uses only the **gradient of the mini-batch** to update the model
- It is fast at first several epochs and becomes **much slower** later.

- SGD with momentum uses a **velocity vector $\boldsymbol{v}$** which **stores the past gradients** together with the **current gradient** to speed up SGD
  - α is a hyper-parameter that indicates how quickly the contributions of previous gradients. In practice, this is usually set to 0.5, 0.9, and 0.99.
  - The momentum primarily solves 2 problems: **poor conditioning** of the Hessian matrix and **variance** in the stochastic gradient.

(Source: Sebastian Ruder)



AdaGrad

AdaGrad

- **<u>Input</u>**: $\eta > 0, \epsilon > 0\ (10^{-6})$, and *initial model* θ

while ***stopping criterion* not met do**

Sample a mini-batch $\{(\boldsymbol{x}^1, \boldsymbol{y}^1), \ldots, (\boldsymbol{x}^b, \boldsymbol{y}^b)\}$

Compute $\boldsymbol{g} = \frac{1}{b}\sum_{i=1}^{b} \nabla_{\boldsymbol{\theta}} l(f(\boldsymbol{x}^i, \boldsymbol{\theta}), \boldsymbol{y}^i)$

Accumulate the square gradient: $\boldsymbol{\gamma} = \boldsymbol{\gamma} + \boldsymbol{g} \odot \boldsymbol{g}$

Update $\boldsymbol{\theta} = \boldsymbol{\theta} - \frac{\boldsymbol{\eta}}{\sqrt{\boldsymbol{\epsilon} + \boldsymbol{\gamma}}} \odot \boldsymbol{g}$

end while

<u>Note</u>: $\odot$ means *element-wise* product

$\boldsymbol{g^1}$	g_1^1	g_2^1	$\ldots$	g_P^1
$\boldsymbol{g^2}$	g_1^2	g_2^2	$\ldots$	g_P^2
$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$
$\boldsymbol{g^t}$	g_1^t	g_2^t	$\ldots$	g_P^t
$\boldsymbol{\gamma}$	$\sum_{i=1}^{t}(g_1^i)^2$	$\sum_{i=1}^{t}(g_2^i)^2$	$\ldots$	$\sum_{i=1}^{t}(g_P^i)^2$

- Learning rates are scaled by the square root of the cumulative sum of squared gradients
- Direction with large partial derivatives
  - Thus, rapid decrease in their learning rates
- Direction with small partial derivatives
  - Hence relatively small decrease in their learning rates
- Weakness: always decrease the learning rate!
  - Excellent for convex problem, but not so good for DL (with non-convex problems)

RMSProp

Not in assessment

RMSProp

- **Input**: $\eta > 0, \epsilon > 0\ (10^{-6}), \beta \in [0,1]$ and *initial model* θ

while *stopping criterion* **not met do**

Sample a mini-batch $\{(\boldsymbol{x}^1, \boldsymbol{y}^1), \dots, (\boldsymbol{x}^b, \boldsymbol{y}^b)\}$

Compute $\boldsymbol{g} = \frac{1}{b}\sum_{i=1}^{b} \nabla_{\boldsymbol{\theta}} l(f(\boldsymbol{x}^i, \boldsymbol{\theta}), \boldsymbol{y}^i)$

Accumulate the square gradient: $\boldsymbol{\gamma} = \boldsymbol{\beta}\boldsymbol{\gamma} + (1 - \boldsymbol{\beta})\boldsymbol{g} \odot \boldsymbol{g}$

Update $\boldsymbol{\theta} = \boldsymbol{\theta} - \frac{\eta}{\sqrt{\epsilon + \gamma}} \odot \boldsymbol{g}$

end while

Note: $\odot$ means element-wise product

- A modification of AdaGrad to work better for **non-convex** setting.
- Instead of cumulative sum, use exponential moving/smoothing average.
- RMSProp has been shown to be an effective and practical optimization algorithm for DNN.
  - Currently one of the go-to optimization methods being employed routinely by DL applications.

Adam

Not in assessment

- The best variant that essentially combines RMSProp with momentum

- Suggested default values: $\eta = 0.001, \alpha_1 = 0.9, \alpha_2 = 0.999$ and $\epsilon = 10^{-6}$.

Adam

- **Input**: $\eta > 0, \epsilon > 0\ (10^{-6}), \beta_1, \beta_2 \in [0,1]$ and *initial model* θ

$\boldsymbol{t = 1}$

while ***stopping criterion*** **not met do**

Sample a mini-batch $\{(\boldsymbol{x}^1, \boldsymbol{y}^1), \ldots, (\boldsymbol{x}^b, \boldsymbol{y}^b)\}$

Compute $\boldsymbol{g} = \frac{1}{b}\sum_{i=1}^{b} \nabla_{\theta} l(f(\boldsymbol{x}^i, \boldsymbol{\theta}), \boldsymbol{y}^i)$

Accumulate the gradient: $\boldsymbol{s} = \boldsymbol{\beta_1 s} + (\boldsymbol{1} - \boldsymbol{\beta_1})\boldsymbol{g}$

Accumulate the square gradient: $\boldsymbol{\gamma} = \boldsymbol{\beta_2 \gamma} + (\boldsymbol{1} - \boldsymbol{\beta_2})\boldsymbol{g} \odot \boldsymbol{g}$

Correct s: $\hat{\boldsymbol{s}} = \frac{\boldsymbol{s}}{1-\beta_1^t}$ # t is the current iteration

Correct γ: $\hat{\boldsymbol{\gamma}} = \frac{\boldsymbol{\gamma}}{1-\beta_2^t}$

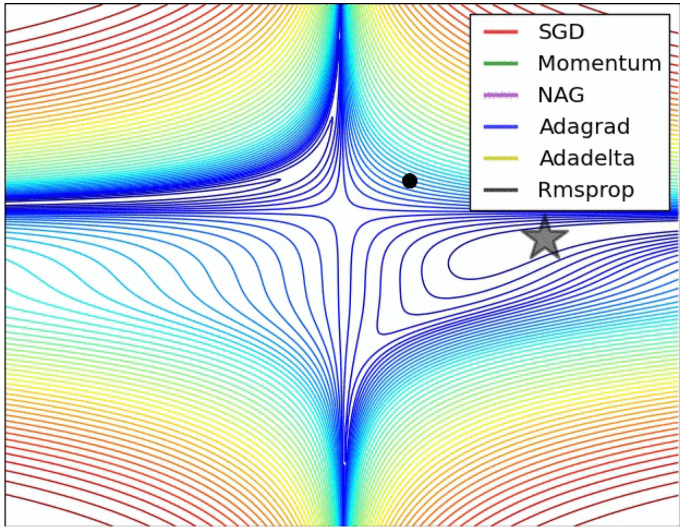
Update $\boldsymbol{\theta} = \boldsymbol{\theta} - \frac{\eta}{\sqrt{\epsilon + \hat{\gamma}}} \odot \hat{\boldsymbol{s}}$

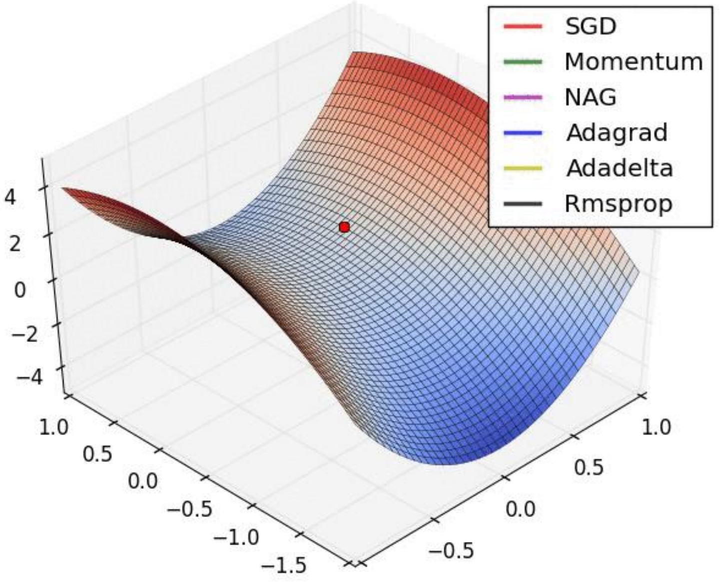
$\boldsymbol{t = t + 1}$

end while

Note: $\odot$ means element-wise product

Visual comparison of all optimizers





[Source: Sebastian Ruder]

Optimizers in deep learning

- **Optimizers**
  - **First order methods**
    - **Adaptive learning rate based on gradients**
      - **Adagrad**

        $$g \leftarrow \nabla J(\theta)$$
        $$\gamma \leftarrow \gamma + g \odot g$$
        $$\theta \leftarrow \theta - \frac{\eta}{\sqrt{\epsilon + \gamma}} \odot g$$

      - **RMSProp**

        $$g \leftarrow \nabla J(\theta)$$
        $$\gamma \leftarrow \beta\gamma + (1-\beta) g \odot g$$
        $$\theta \leftarrow \theta - \frac{\eta}{\sqrt{\epsilon + \gamma}} \odot g$$

      - **Adam**

        $$t \leftarrow t + 1$$
        $$g \leftarrow \nabla J(\theta)$$
        $$s \leftarrow \beta_1 s + (1-\beta_1) g$$
        $$\gamma \leftarrow \beta_2 \gamma + (1-\beta_2) g \odot g$$
        $$\hat{s} \leftarrow \frac{s}{1-\beta_1^t}$$
        $$\hat{\gamma} \leftarrow \frac{\gamma}{1-\beta_2^t}$$
        $$\theta \leftarrow \theta - \eta \frac{\hat{s}}{\sqrt{\epsilon + \hat{\gamma}}}$$

    - **Standard Mini-batched SGD**
      - **SGD**

        $$\theta \leftarrow \theta - \eta \nabla J(\theta)$$

      - **SGD with momentum**

        $$v \leftarrow \alpha v + (1-\alpha) \nabla J(\theta)$$
        $$\theta \leftarrow \theta - \eta v$$

  - **Second order methods (Newton method)**

    $$\theta \leftarrow \theta - \eta H^{-1} g$$

    - **Impossible** for DL

g^1	g_1^1	g_2^1	…	g_P^1
g^2	g_1^2	g_2^2	…	g_P^2
…	…	…	…	…
g^t	g_1^t	g_2^t	…	g_P^t
γ	$\sum_{i=1}^{t} (g_1^i)^2$	$\sum_{i=1}^{t} (g_2^i)^2$	…	$\sum_{i=1}^{t} (g_P^i)^2$

Summary

- Optimization problem in DL and ML
  - Regularization term + Empirical loss term
- Gradient descent
- Stochastic gradient descent
- Backward propagation
- Other optimizers in DL
  - SGD with momentum, Adagrad, RMSProp, and Adam
- First order methods and second order methods

Thanks for your attention!