

# ECE 598NSG/498NSU

## Deep Learning in Hardware

### Fall 2020

## Training DNNs – SGD, Backprop and its Variants

Naresh Shanbhag

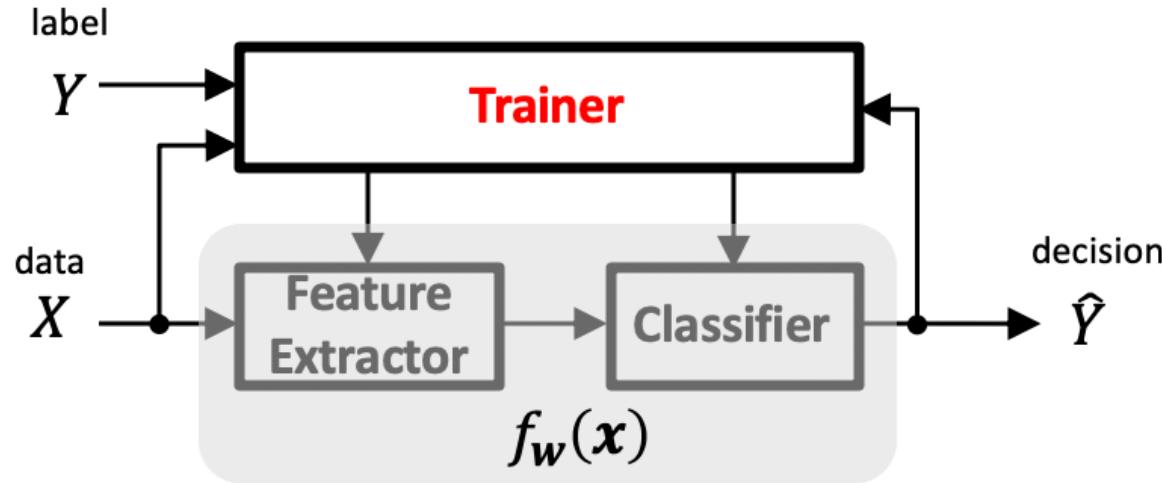
Department of Electrical and Computer Engineering  
University of Illinois at Urbana-Champaign

<http://shanbhag.ece.uiuc.edu>

# Today

- Overview of DNN training – the big picture
- The Stochastic Gradient Descent (SGD) training algorithm
- The back-prop algorithm

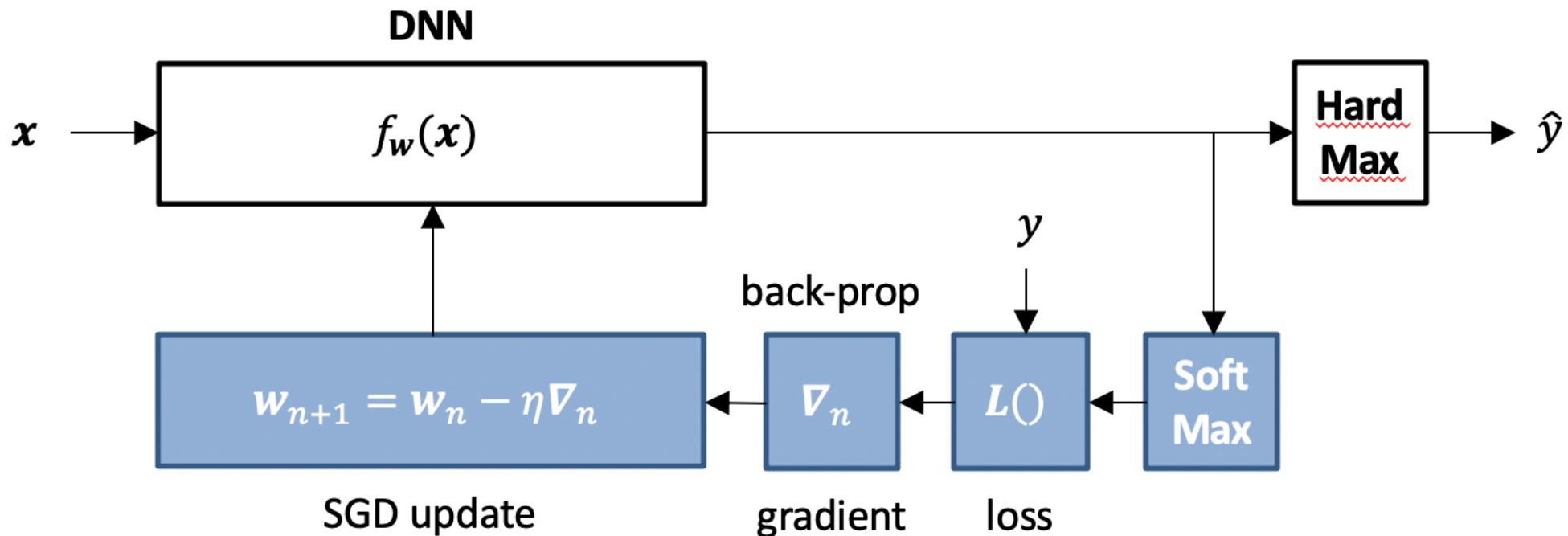
# Supervised Training



- **Sample:**  $z = (x, y) = (\text{data}, \text{label})$ ;
- **Training sample (example):** samples used in training.
- **Loss function:**  $L(\hat{y} = f_w(x), y)$ ; sample-specific value
- Family of functions parametrized by  $w$ :  $\mathcal{F}$ :  $f_w(x) = \hat{y}$
- Loss function evaluated on the training set:  $Q(w)$
- **learning (convergence) curves; learning rate; stability**

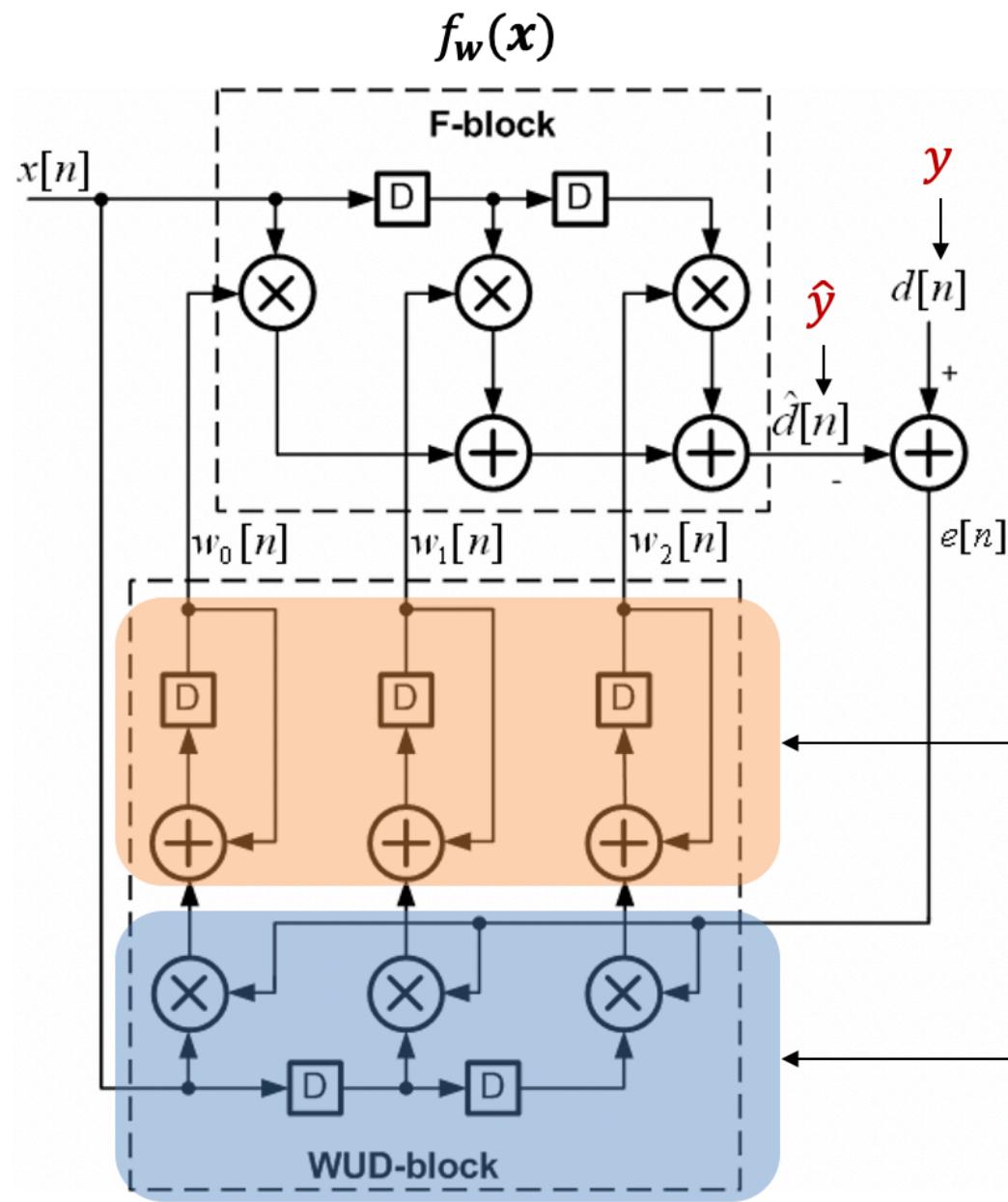
# DNN Training Set-up

# Overview

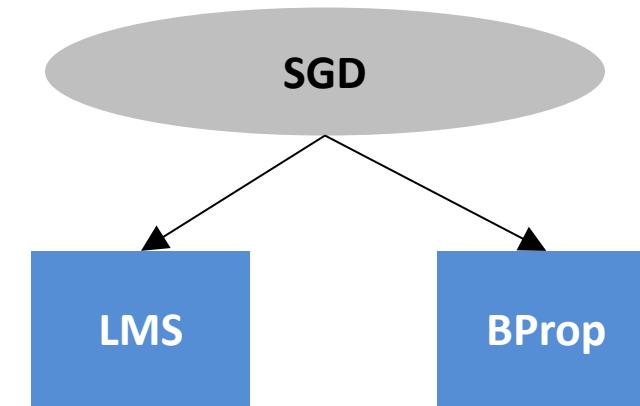
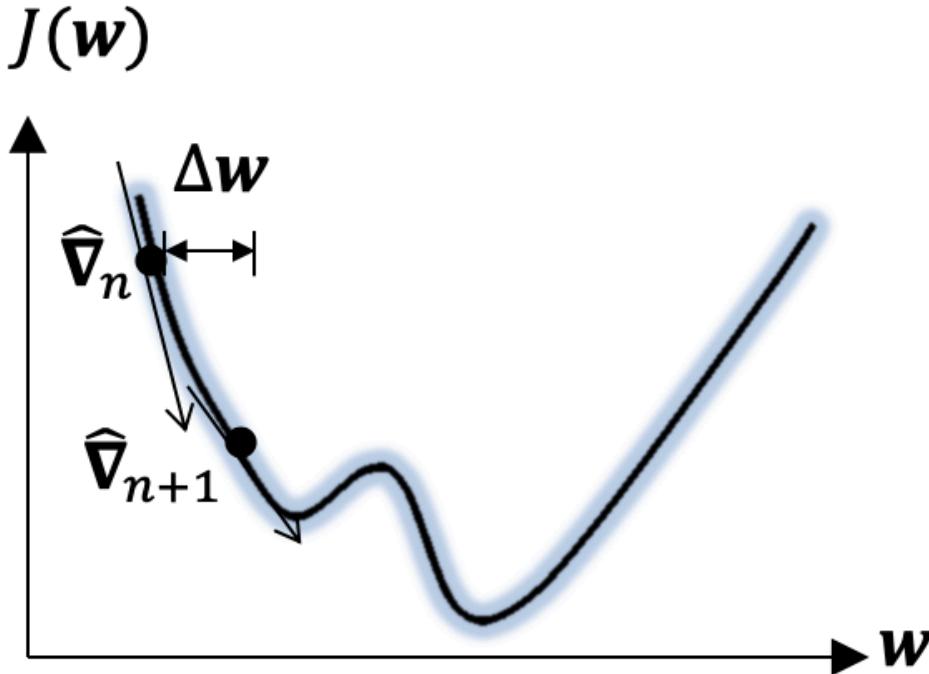


- Forward pass – generates predicted label  $\hat{y}$
- Backward pass – SGD-based back-prop to update weights

# LMS via DNN Lens



# Stochastic Gradient Descent

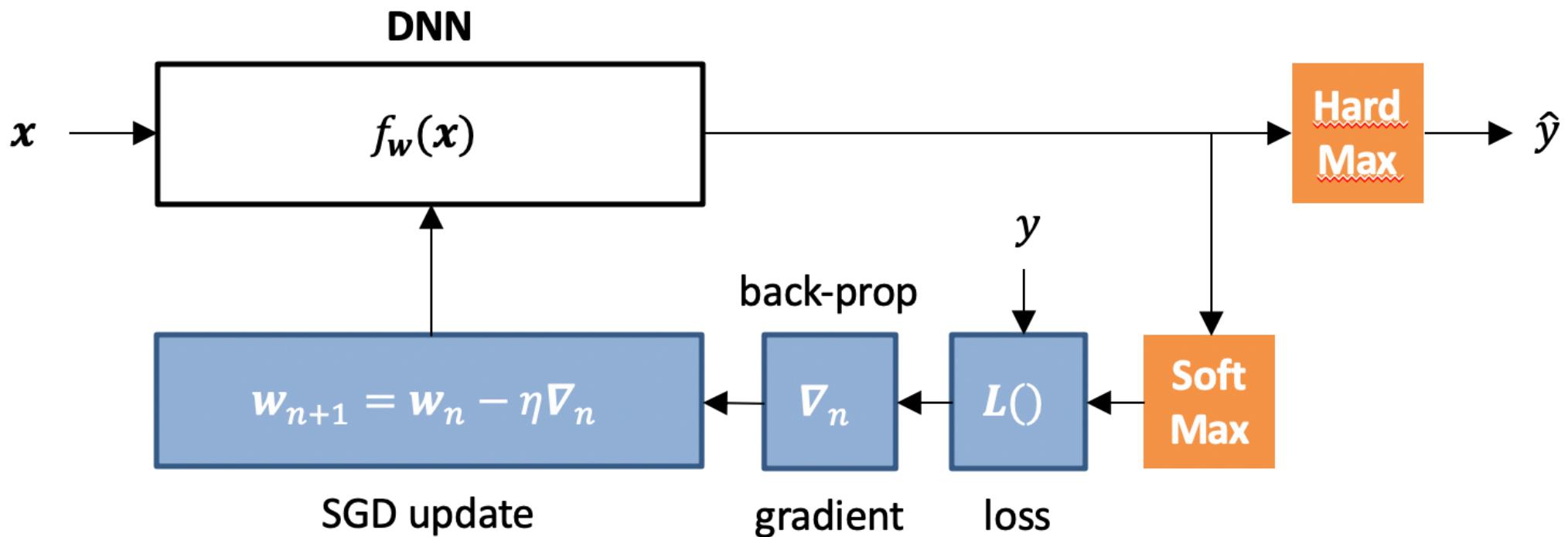


- Calculate instantaneous gradient of loss function; update weight parameter by adding a small increment in the negative gradient

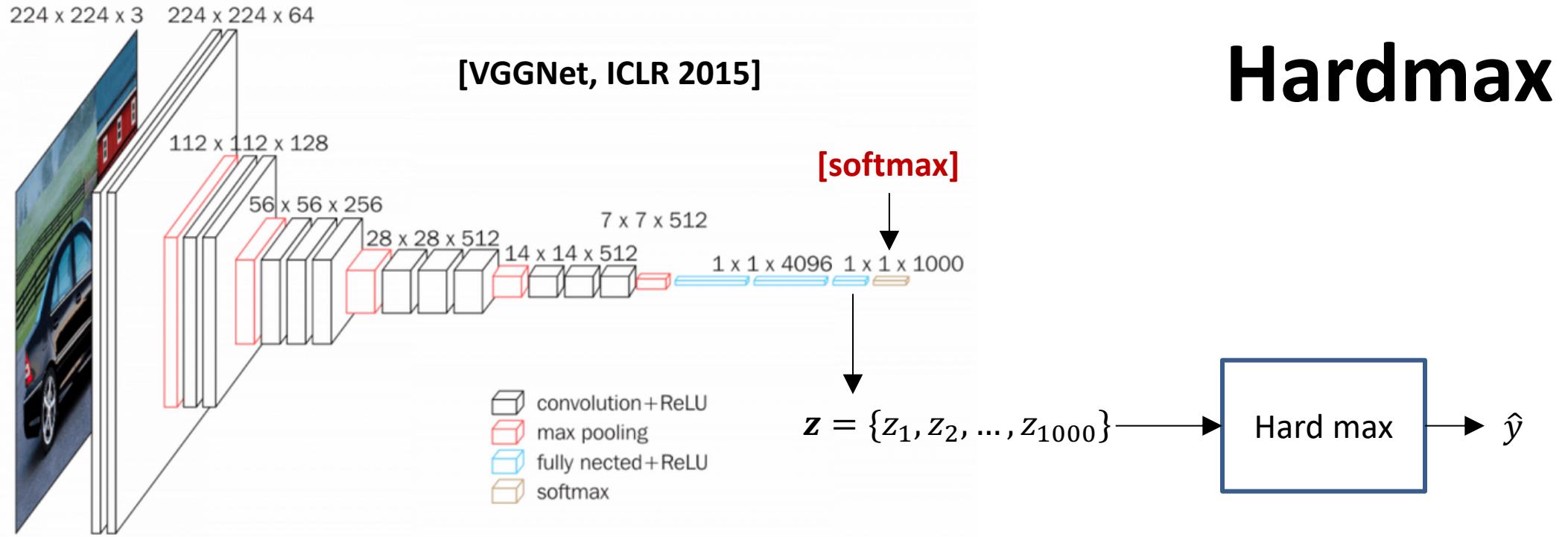
$$w_{n+1} = w_n + \mu(-\hat{\nabla}_n)$$

- $\hat{\nabla}_n = \frac{\partial \hat{J}(w_n)}{\partial w_n}$   $\Rightarrow$  is the gradient of the *instantaneous loss function* wrt  $w_n$
- $\mu$ : step-size or learning rate

# Overview



- Forward pass – generates predicted label  $\hat{y}$
- Backward pass – SGD-based back-prop to update weights

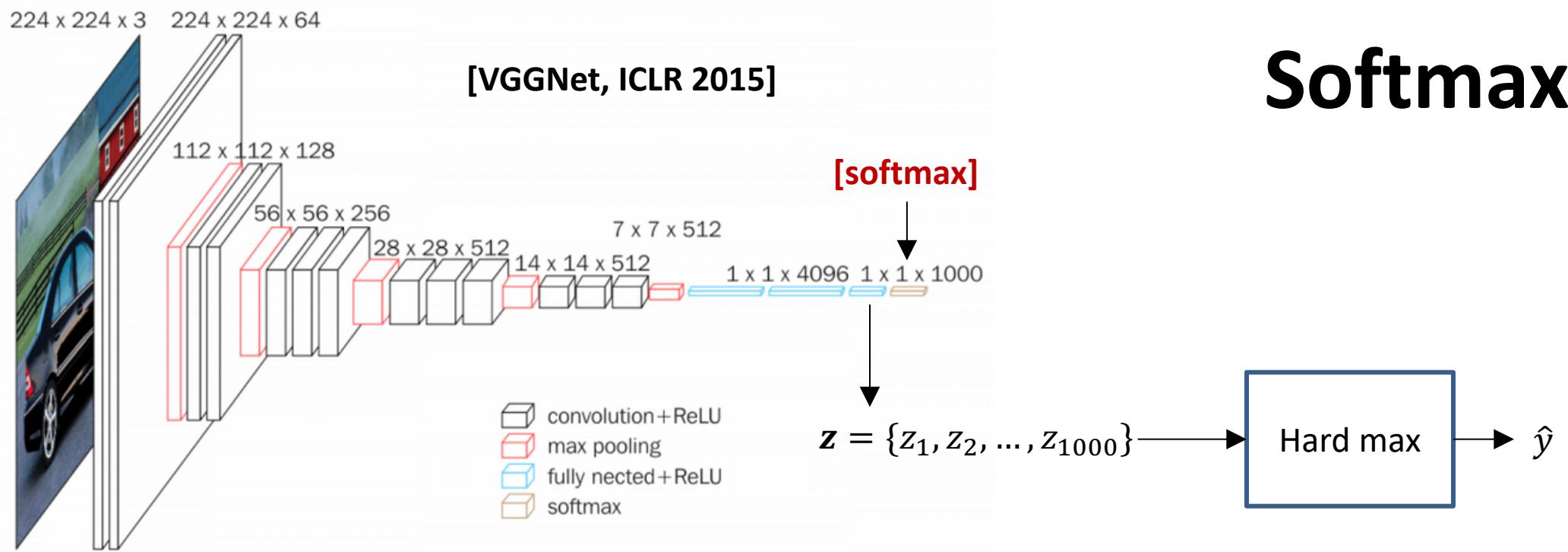


- $N$ -way hard max: outputs are final decisions (Top-1, Top-5):

$$\hat{y} = \max(z_1, z_2, \dots, z_N) \quad \text{or} \quad \hat{y} = \max(\sigma_1, \sigma_2, \dots, \sigma_N)$$

- $z_i$ : last FC layer outputs;  $\sigma_i$ : softmax outputs;

# Softmax

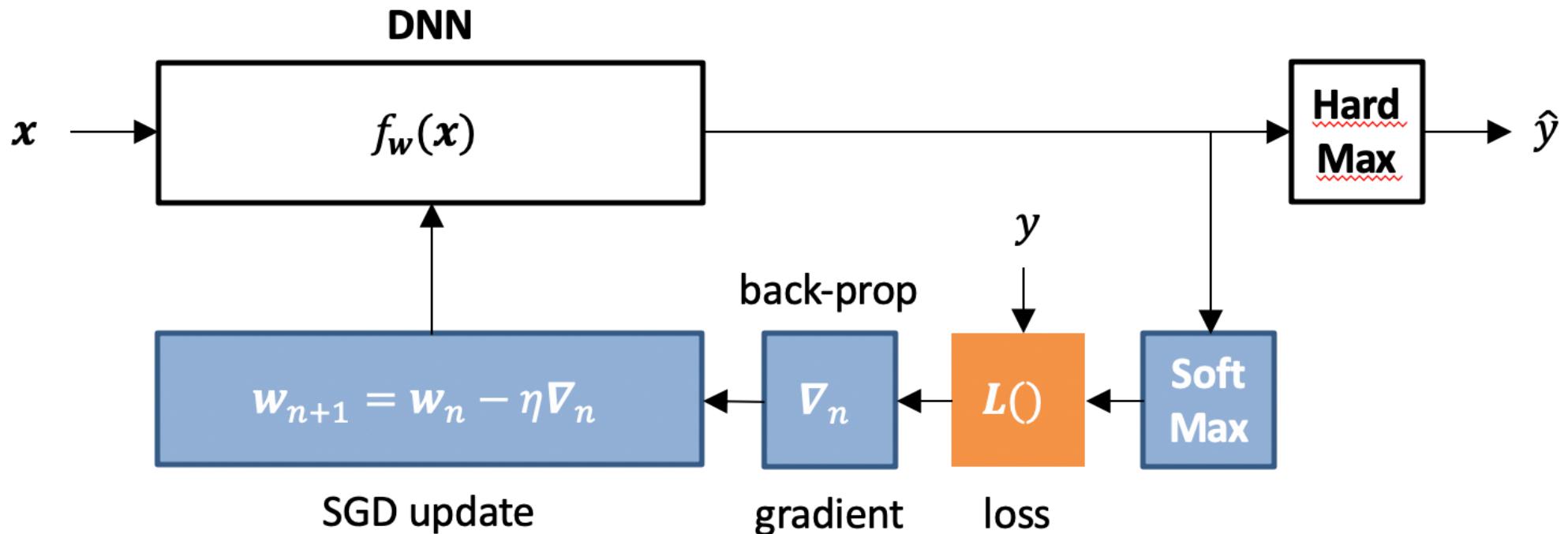


- $N$ -way soft max outputs estimated class probabilities: (used for loss function computation)
- E.g.: binary case ( $N = 2$ ):

$$\sigma_i = \frac{e^{z_i}}{\sum_{k=1}^N e^{z_k}}$$

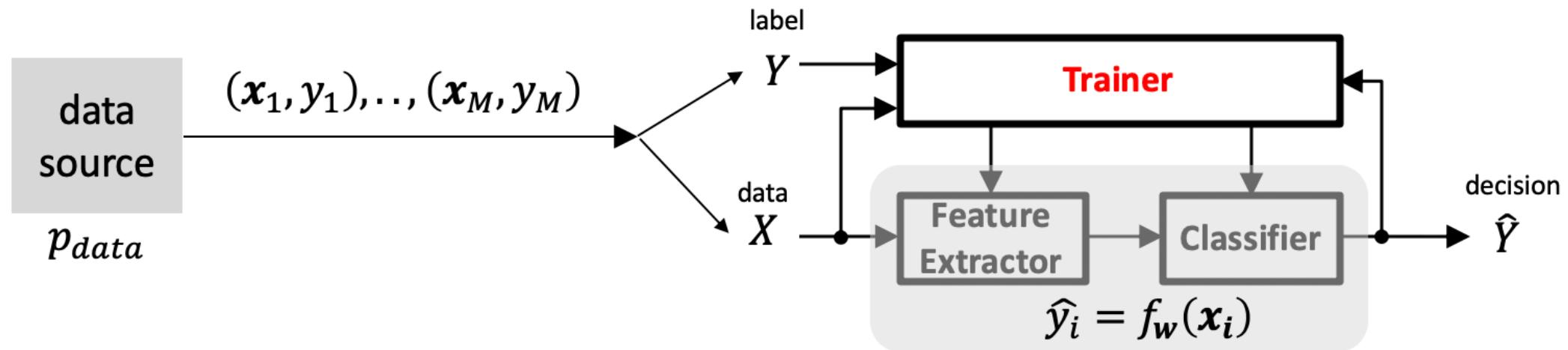
$$\sigma_1 = \frac{e^{z_1}}{e^{z_1} + e^{z_2}}; \quad \sigma_2 = \frac{e^{z_2}}{e^{z_1} + e^{z_2}}$$

# Overview



- Forward pass – generates predicted label  $\hat{y}$
- Backward pass – SGD-based back-prop to update weights

# Loss Function

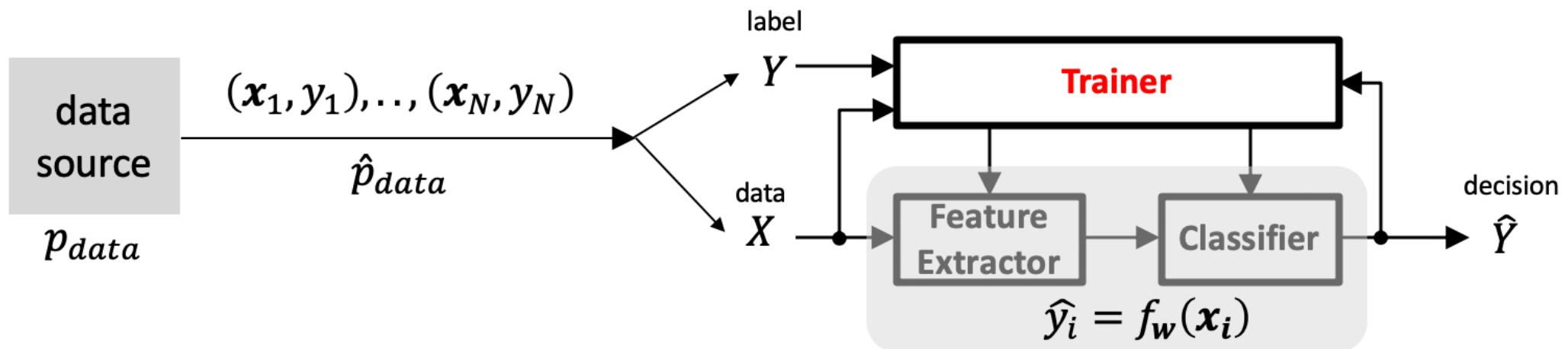


- measures generalization performance

$$J(\mathbf{w}) = \mathbb{E}_{\mathbf{x} \sim p_{data}} [L(f_{\mathbf{w}}(\mathbf{x}), y)] = \mathbb{E}_{\mathbf{x} \sim p_{data}} [L(\hat{y}, y)]$$

expectation is over the (**unknown**) data generating distribution  $p_{data}$ .

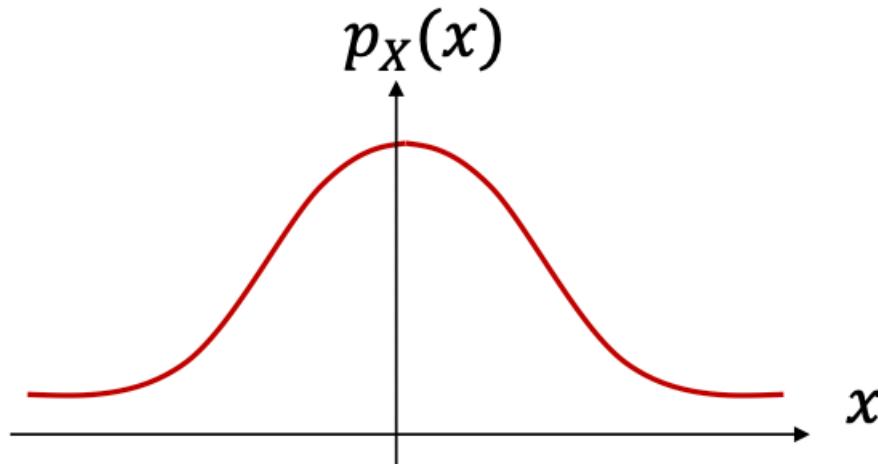
# (Empirical) Loss Function



- True data distribution is unknown → empirical (sample) estimate of loss
- sample averaged loss over the **training set**.

$$\hat{J}(\mathbf{w}) = \frac{1}{N} \sum_{i=1}^N L(f_{\mathbf{w}}(\mathbf{x}_i), y_i) = \mathbb{E}_{\mathbf{x} \sim \hat{p}_{data}} [L(f_{\mathbf{w}}(\mathbf{x}), y)]$$

# Shannon Entropy



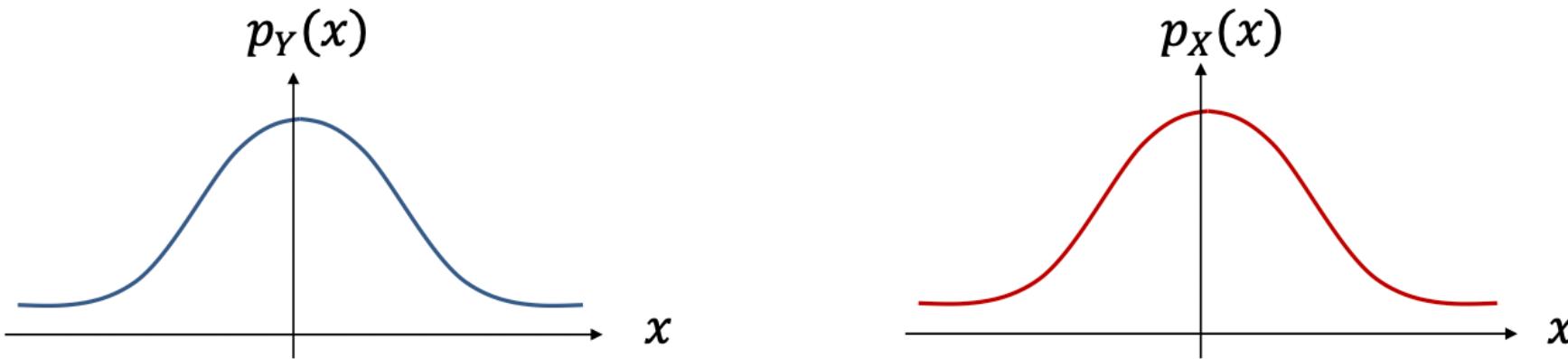
- Quantifies the uncertainty (information content) in a RV  $x$

$$H(X) = -\sum_x p_X(x) \log_2(p_X(x)) = -\mathbb{E}_{\sim p_X} [\log_2(p_X(x))] \text{ (in bits)}$$

- If  $X \in \mathcal{F}$  is a uniformly distributed discrete RV then

$$H(X) = \log_2 |\mathcal{F}| \text{ bits}$$

# Kullback-Leibler Divergence



- Distance between two distributions  $p_X(x)$  &  $p_Y(x)$  defined over the same RV  $x$ :

$$\begin{aligned} D_{KL}(p_X || p_Y) &= \mathbb{E}_{\sim p_X} \left[ \log_2 \left( \frac{p_X(x)}{p_Y(x)} \right) \right] \\ &= \mathbb{E}_{\sim p_X} [\log_2(p_X(x))] - \mathbb{E}_{\sim p_X} [\log_2(p_Y(x))] = -H(X) + H(X, Y) \end{aligned}$$

- $D_{KL}(p_X || p_Y) = 0$  iff  $p_X(x) = p_Y(x)$
- can be used to compare two probability distributions

cross-entropy

# Cross-entropy (Loss)



- Cross-entropy:  $H(X, Y) = - \sum_x p_X(x) \log_2(p_Y(x))$
- minimizing  $D_{KL}(p_X || p_Y)$   $\rightarrow$  minimizing  $H(X, Y)$
- Cross-entropy loss  $\rightarrow$  differential function of its parameters

$$J(\mathbf{w}) = - \sum_i y_i \log_2(\sigma_i)$$

$$\sigma_i = \frac{e^{z_i}}{\sum_{k=1}^N e^{z_k}};$$

$y_i$ : 1-hot encoded true class labels

# Loss Function Example - MSE

- $L(f_w(\mathbf{x}_i), y_i) = L(\hat{y}_i, y_i) = (\hat{y}_i - y_i)^2$
- true average loss:

$$\begin{aligned} J(\mathbf{w}) &= \mathbb{E}_{\mathbf{x} \sim p_{data}} [L(f_w(\mathbf{x}), y)] \\ &= \sum_{\mathbf{x}} (f_w(\mathbf{x}) - y)^2 p_{data}(\mathbf{x}, y) \end{aligned}$$

- sample average loss

$$\begin{aligned} \hat{J}(\mathbf{w}) &= \mathbb{E}_{\mathbf{x} \sim \hat{p}_{data}} [L(f_w(\mathbf{x}), y)] \\ &= \frac{1}{N} \sum_{i=1}^N (f_w(\mathbf{x}) - y_i)^2 \end{aligned}$$

# Regularization

- DNNs are over-parameterized → many solutions possible
- regularization guides the choice of a certain type of solution → intended to reduce generalization error but not training error
- includes additional constraints on parameter values
- many ways to regularize – norm-based, drop-out, soft weight sharing....
- Example:  $L_2$  regularization (weight decay)

$$\tilde{J}(\mathbf{w}) = \frac{\alpha}{2} \mathbf{w}^T \mathbf{w} + J(\mathbf{w}; \mathbf{x}, y)$$



$$\mathbf{w}_{n+1} = (1 - \eta\alpha) \mathbf{w}_n - \eta \nabla_{\mathbf{w}} J(\mathbf{w}; \mathbf{x}, y)$$

# History

- SGD applied to a DNN (multi-layer network) is called **back-propagation** algorithm
- Based on the application of chain rule and SGD
- First proposed in Paul Werbos's doctoral dissertation [1974]

P. Werbos, "Beyond regression: New tools for prediction and analysis in the behavioral sciences," Ph.D. dissertation, Committee on Appl. Math., Harvard Univ., Cambridge, MA, Nov. 1974.

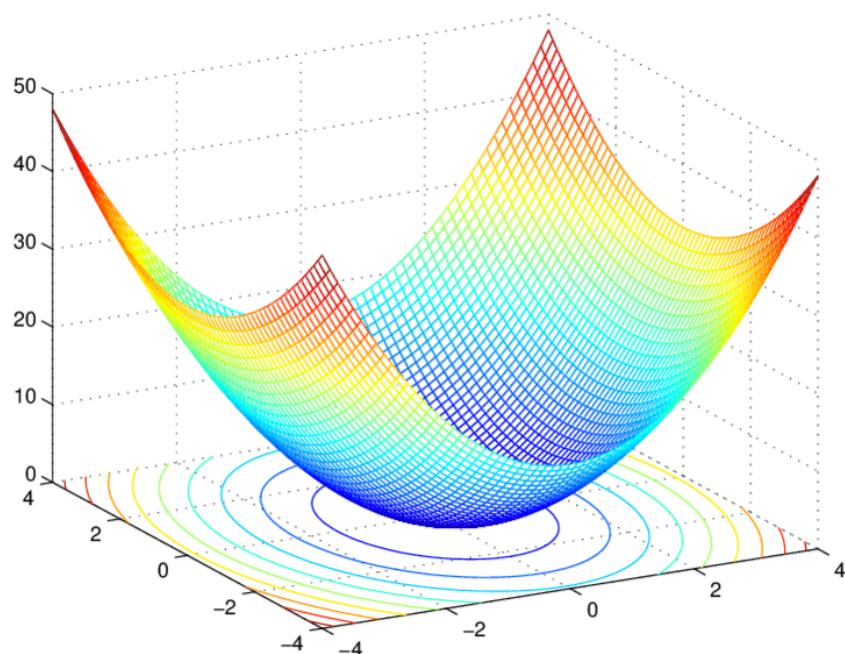
- Rediscovered and popularized by Rumelhart, Hinton, and Williams [1986]

D. Rumelhart, D. Hinton, and G. Williams, "Learning internal representations by error propagation," in D. Rumelhart and F. McClelland, eds., *Parallel Distributed Processing*, Vol. 1. Cambridge, MA: M.I.T. Press, 1986.

# Loss Landscape of DNNs

---

## Linear regressor



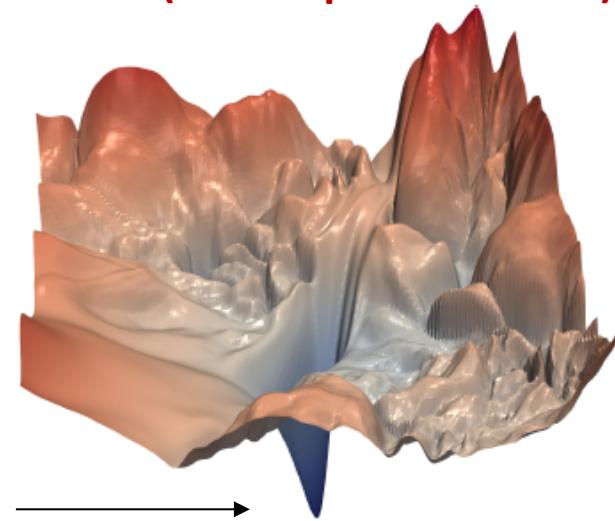
## Visualizing the Loss Landscape of Neural Nets

---

Hao Li<sup>1</sup>, Zheng Xu<sup>1</sup>, Gavin Taylor<sup>2</sup>, Christoph Studer<sup>3</sup>, Tom Goldstein<sup>1</sup>

<sup>1</sup>University of Maryland, College Park <sup>2</sup>United States Naval Academy <sup>3</sup>Cornell University  
`{haoli,xuzh,tomg}@cs.umd.edu, taylor@usna.edu, studer@cornell.edu`

## ResNet-56 (w.o. skip connections)



How to find this? →

- DNNs have non-convex loss function → loss landscape has many peaks and valleys



drone  $X_i = (x_i, y_i)$ : example

$x_i$        $y_i$

$X_1, X_2, \dots, X_M$

random shuffle

$X'_1, X'_2, \dots, X'_M$

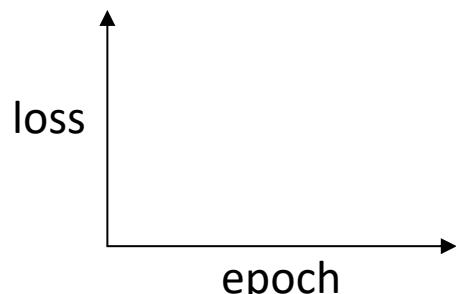
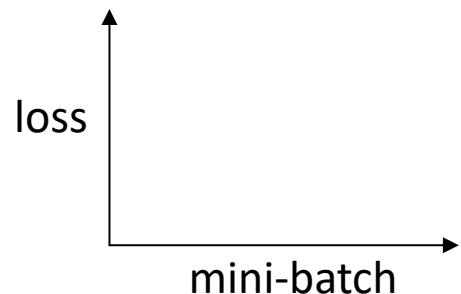
training set

$[X'_1, X'_2, \dots, X'_B]$     $[X'_{B+1}, X'_{B+2}, \dots, X'_{2B}] \dots [X'_{M-B-1}, X'_{M-B}, \dots, X'_M]$    epoch

mini-batch

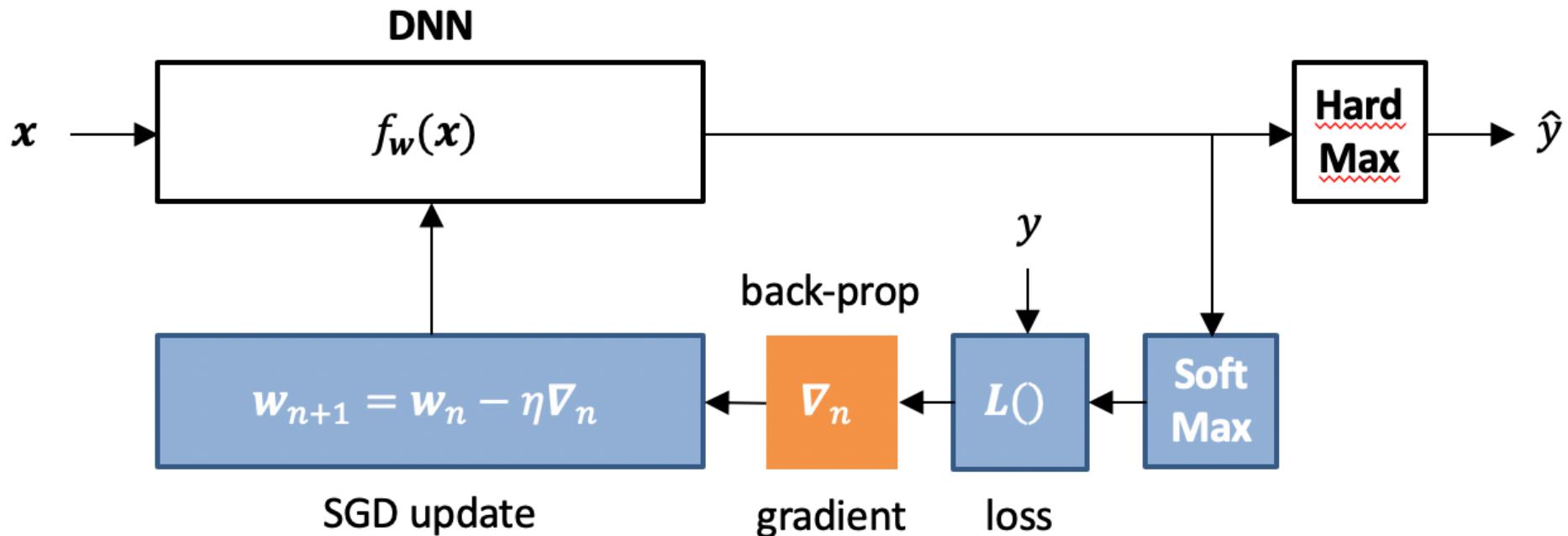
mini-batch

mini-batch



# The Back Propagation Algorithm

# Overview



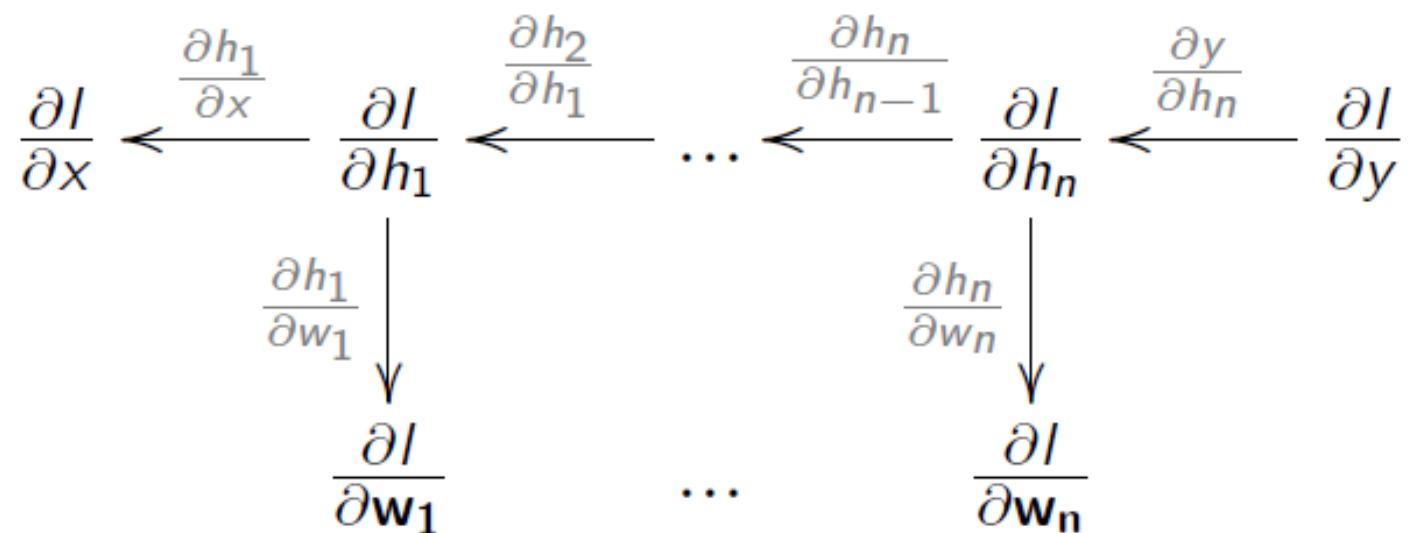
- Forward pass – generates predicted label  $\hat{y}$
- Backward pass – SGD-based back-prop to update weights

# Backpropagation

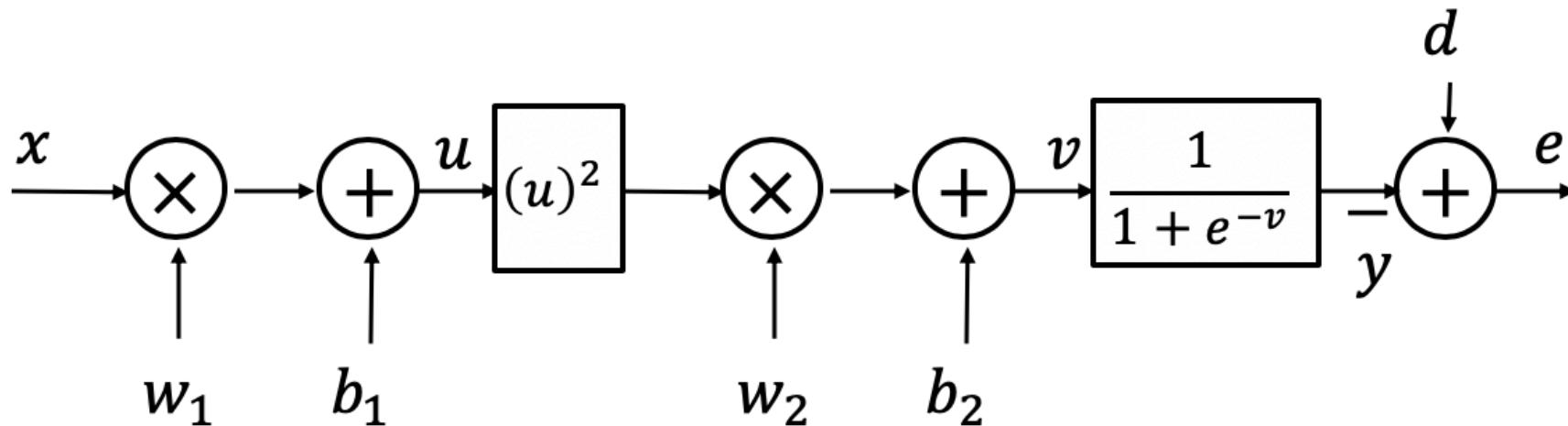
- determine  $\mathbf{w}$  that minimizes loss function (empirical risk):

$$\frac{1}{n} \sum_1^n l(h(\mathbf{x}_i; \mathbf{w}), y_i)$$

- backpropagation = use of chain rule

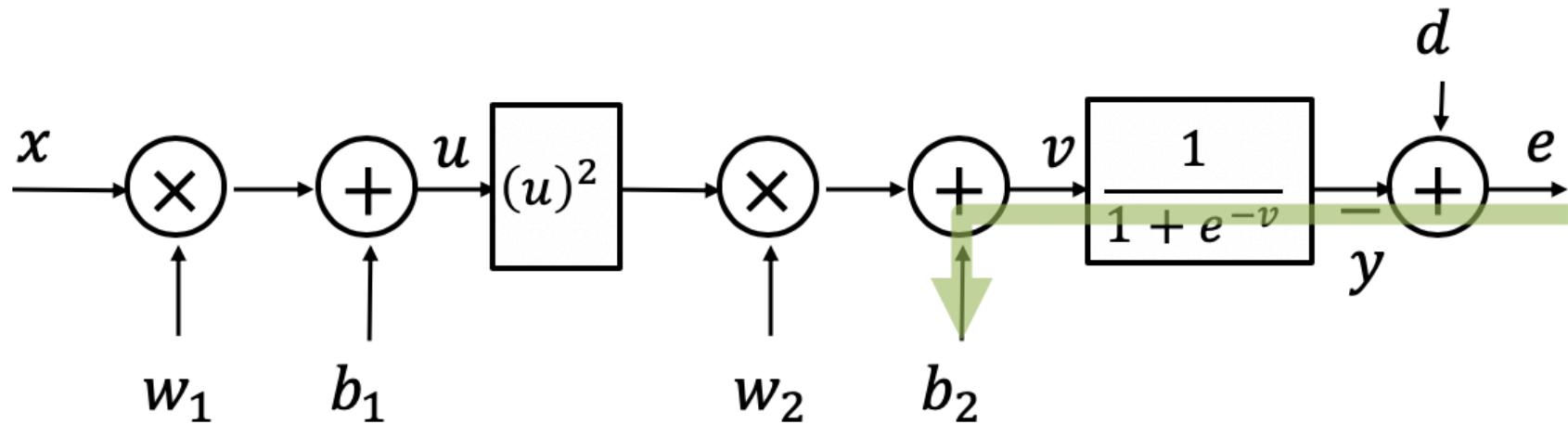


# Optimization with a Non-Linear System



- much like in LMS, we would like to use SGD in order to learn values of  $w_1, b_1, w_2, b_2$  to minimize  $E[e^2]$
- need to find:  $\frac{\partial e^2}{\partial w_1}, \frac{\partial e^2}{\partial b_1}, \frac{\partial e^2}{\partial w_2}, \frac{\partial e^2}{\partial b_2}$
- to do so: use back-propagation (or chain rule)

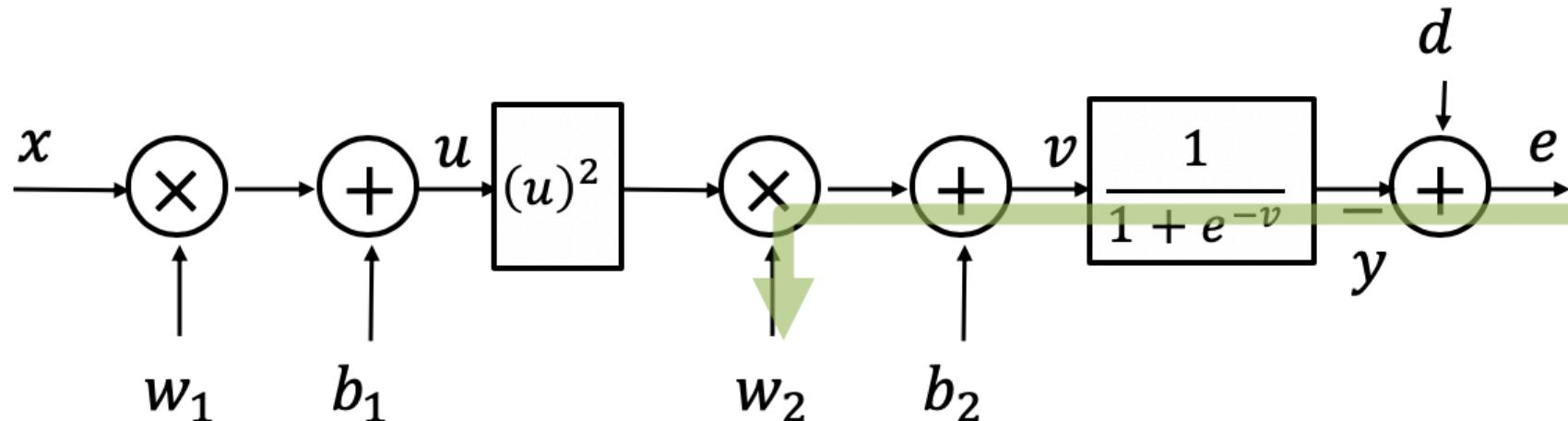
# Back-propagating to $b_2$



- $$\frac{\partial e^2}{\partial b_2} = \frac{\partial e^2}{\partial y} \cdot \frac{\partial y}{\partial v} \cdot \frac{\partial v}{\partial b_2} = -2ey(1 - y)$$

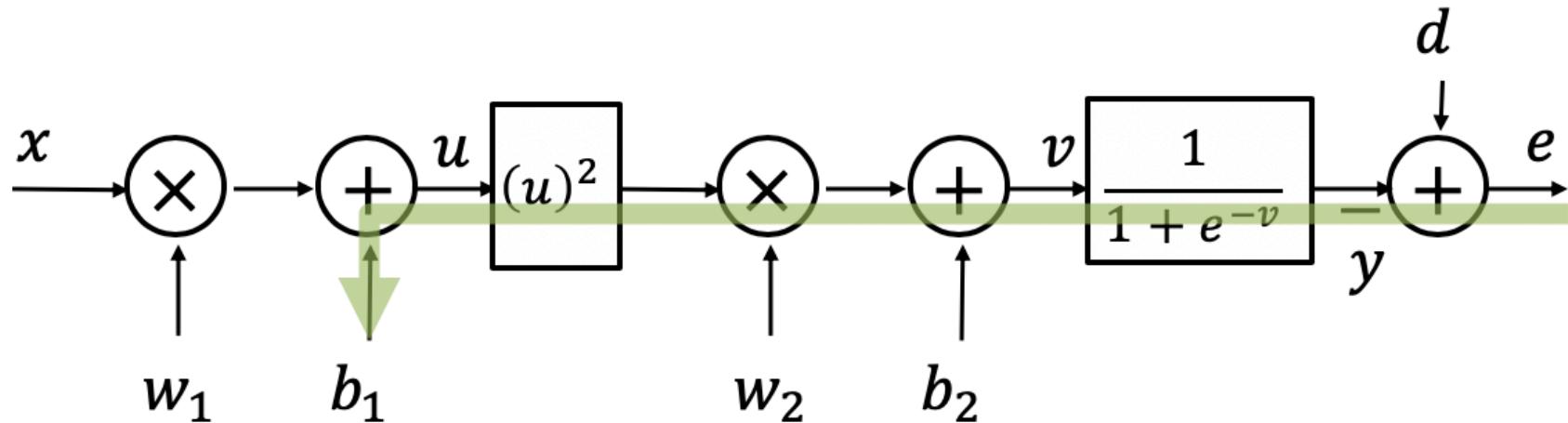
$-2e$   
 $y(1 - y)$   
 $1$
- SGD update on  $b_2$ :  $b_{2,t+1} \leftarrow b_{2,t} + 2\mu e_t y_t (1 - y_t)$

# Back-propagating to $w_2$



- $$\frac{\partial e^2}{\partial w_2} = \frac{\partial e^2}{\partial y} \cdot \frac{\partial y}{\partial v} \cdot \frac{\partial v}{\partial w_2} = -2ey(1 - y)u^2$$
  
-2e  
 $y(1 - y)$   
 $u^2$
- SGD update on  $w_2$ :  $w_{2,t+1} \leftarrow w_{2,t} + 2\mu e_t y_t (1 - y_t) u_t^2$

# Back-propagating to $b_1$

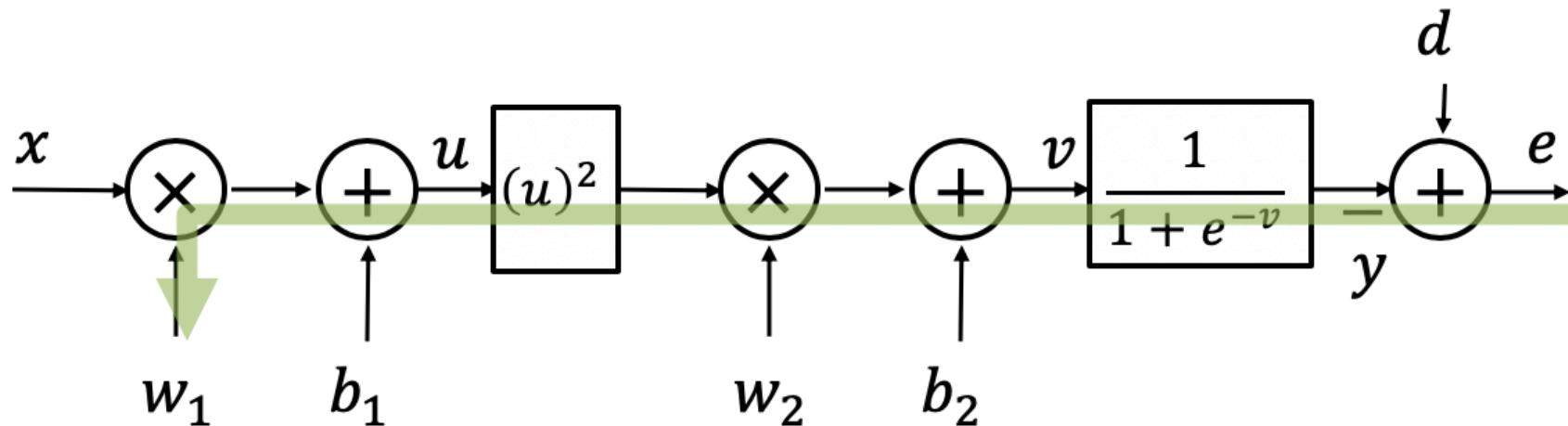


$$\bullet \frac{\partial e^2}{\partial b_1} = \frac{\partial e^2}{\partial y} \cdot \frac{\partial y}{\partial v} \cdot \frac{\partial v}{\partial u} \cdot \frac{\partial u}{\partial b_1} = -4ey(1 - y)uw_2$$

$$\begin{matrix} -2e \\ y(1 - y) \\ 2uw_2 \\ 1 \end{matrix}$$

$$\bullet \text{SGD update on } b_1: b_{1,t+1} \leftarrow b_{1,t} + 4\mu e_t y_t (1 - y_t) u_t w_{2,t}$$

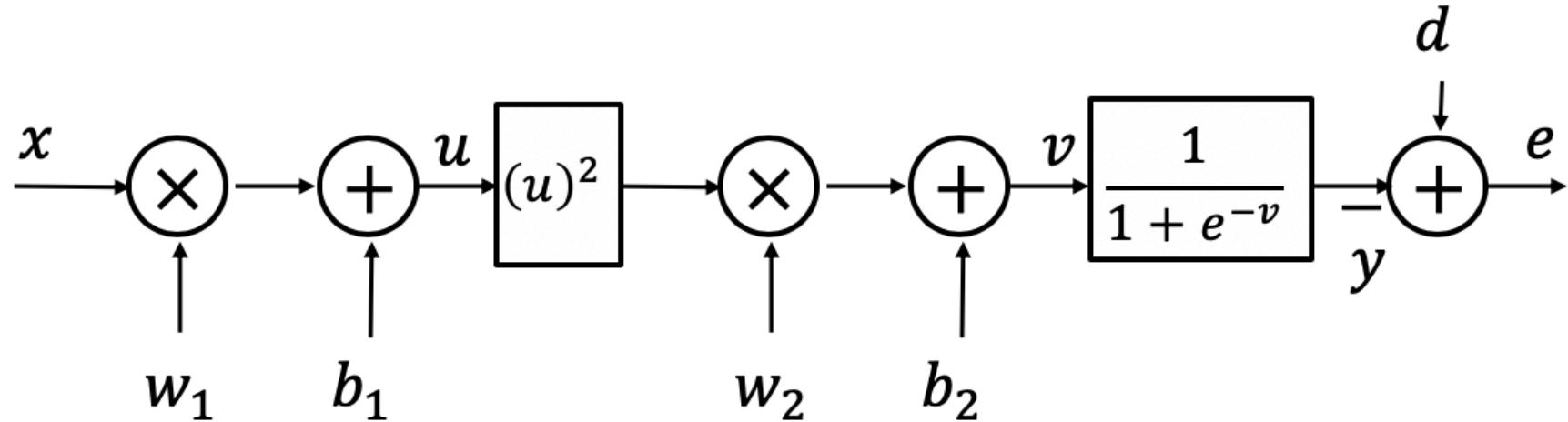
# Back-propagating to $w_1$



- $$\frac{\partial e^2}{\partial w_1} = \frac{\partial e^2}{\partial y} \cdot \frac{\partial y}{\partial v} \cdot \frac{\partial v}{\partial u} \cdot \frac{\partial u}{\partial w_1} = -4ey(1-y)uw_2x$$

$\downarrow$   $\downarrow$   $\downarrow$   $\downarrow$   
 $-2e$   $y(1-y)$   $2uw_2$   $x$
- SGD update on  $w_1$ :  $w_{1,t+1} \leftarrow w_{1,t} + 4\mu e_t y_t (1 - y_t) u_t w_{2,t} x_t$

# Summary



$$w_{1,t+1} \leftarrow w_{1,t} + 4\mu e_t y_t (1 - y_t) u_t w_{2,t} x_t$$

$$b_{1,t+1} \leftarrow b_{1,t} + 4\mu e_t y_t (1 - y_t) u_t w_{2,t}$$

$$w_{2,t+1} \leftarrow w_{2,t} + 2\mu e_t y_t (1 - y_t) u_t^2$$

$$b_{2,t+1} \leftarrow b_{2,t} + 2\mu e_t y_t (1 - y_t)$$

# Gradient Computation in PyTorch

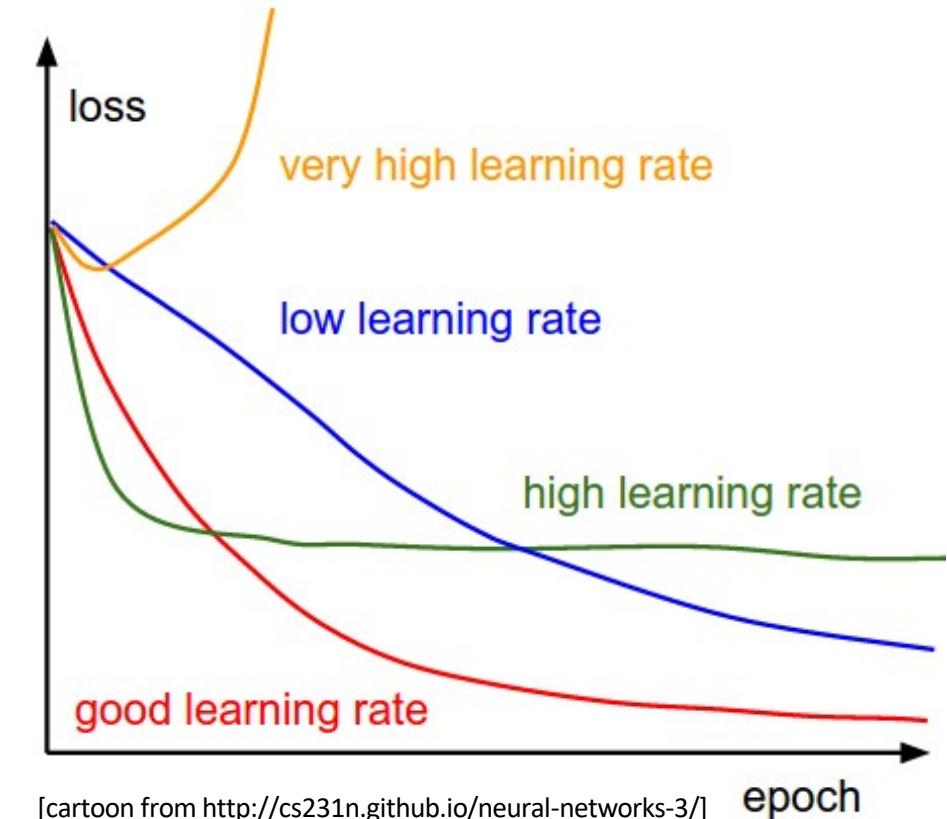
- deep learning packages provide a tool which allows you to compute derivatives effortlessly
- every computation graph is stored and its dual is computed
- the dual graph is used to compute derivatives automatically

**using SGD with  
Pytorch: a few  
lines**

```
outputs = net(inputs) → feedforward
loss = criterion(outputs, targets)
loss.backward() → automatic differentiation
with torch.no_grad():
    for param in net.parameters():
        param.data.add_(-lr, param.grad.data) → weight update
        param.grad.data.zero_()
```

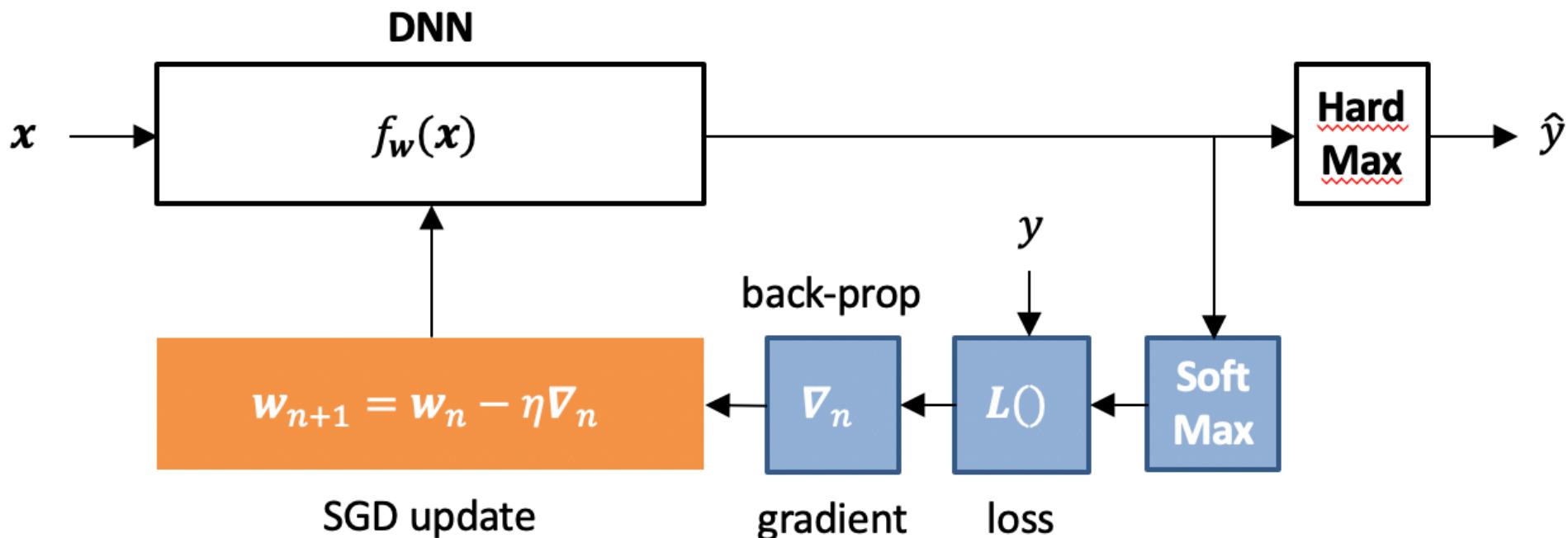
# Impact of Learning Rate on Convergence

- a **very large** learning rate → instability
- a **large learning rate** wiggles at the minimum loss point
- a **small learning rate** causes slow convergence
- the learning rate should be neither too small nor too large
- gear shifting is a very popular trick to obtain the best of both worlds



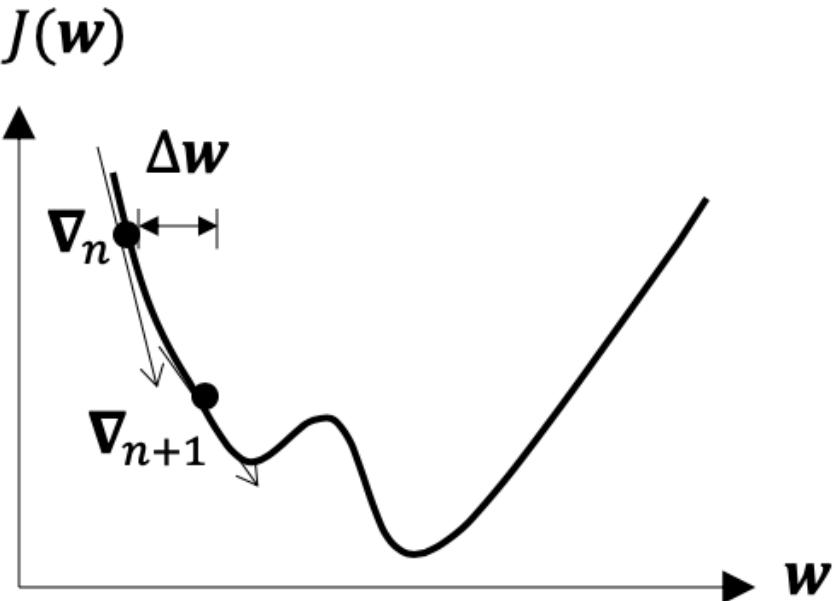
# The Stochastic Gradient Descent (SGD) Algorithm

# Overview



- Forward pass – generates predicted label  $\hat{y}$
- Backward pass – SGD-based back-prop to update weights

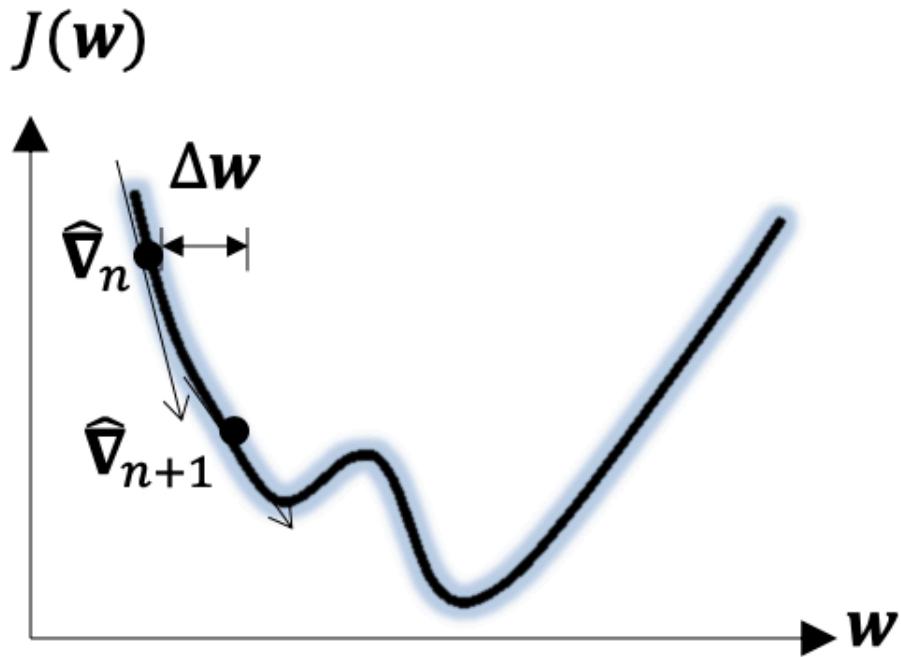
# Gradient Descent



$$w_{n+1} = w_n + \mu(-\nabla_n)$$

- $\nabla_n = \frac{\partial J(w)}{\partial w} |_{w=w_n} \Rightarrow$  is the gradient of the true loss function wrt  $w$  evaluated at  $w = w_n$
- $\mu$ : step-size or **learning rate**
- Loss function reduces if it is smooth. Can get stuck in a local minima.
- Gear-shift  $\mu$  to jump over local minima.

# Stochastic Gradient Descent

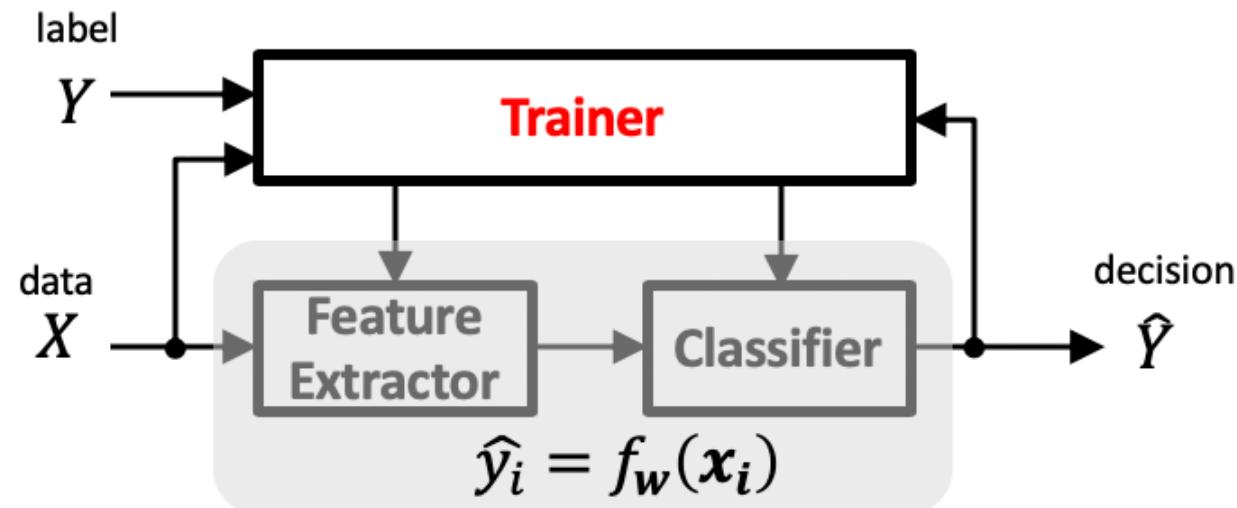


Replaces true gradient  $\nabla_n$  with sample gradient  $\widehat{\nabla}_n$

$$w_{n+1} = w_n + \mu(-\widehat{\nabla}_n)$$

- $\widehat{\nabla}_n = \frac{\partial \hat{J}(w_n)}{\partial w_n}$   $\Rightarrow$  is the gradient of the *sample loss function* wrt  $w_n$
- $\mu$ : step-size or **learning rate**

# SGD-based Learning Algorithms



- choose a **predictor function**:  $\hat{y}_i = f_w(x_i)$  ( $w$  is unknown)
- choose a **loss function**:  $L(\hat{y} = f_w(x), y)$  (differentiable)
- choose a data set:  $(x_i, y_i)$ ; initialize weights:  $w = w_0$
- compute sample gradient of loss function:  $\hat{\nabla}_i$
- determine update rule:  $w_{i+1} = w_i + \mu(-\hat{\nabla}_i)$

# History

- SGD applied to a DNN (multi-layer network) results in **back-propagation** algorithm
- Based on the application of chain rule and SGD
- First proposed in Paul Werbos's doctoral dissertation [1974]

P. Werbos, "Beyond regression: New tools for prediction and analysis in the behavioral sciences," Ph.D. dissertation, Committee on Appl. Math., Harvard Univ., Cambridge, MA, Nov. 1974.

- Rediscovered and popularized by Rumelhart, Hinton, and Williams [1986]

D. Rumelhart, D. Hinton, and G. Williams, "Learning internal representations by error propagation," in D. Rumelhart and F. McClelland, eds., *Parallel Distributed Processing, Vol. 1*. Cambridge, MA: M.I.T. Press, 1986.

# Example - SVM

$$\mathbf{w}_{n+1} = \mathbf{w}_n - \gamma \hat{\nabla}_n$$

- sample loss function:

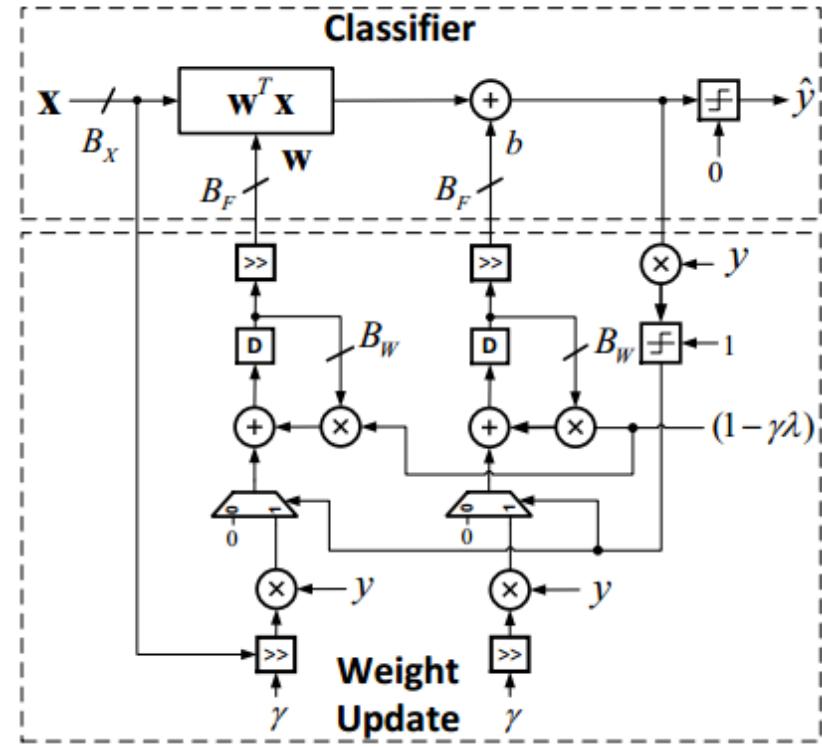
$$\hat{J}(\mathbf{w}_n) = \frac{1}{2}\lambda\|\mathbf{w}_n\|^2 + \max(0, 1 - y_n \mathbf{w}_n^T \mathbf{x}_n)$$

- sample gradient:

$$\hat{\nabla}_n = \frac{\partial \hat{J}(\mathbf{w}_n)}{\partial \mathbf{w}_n} = \lambda \mathbf{w}_n + \begin{cases} 0 & \text{if } y_n \mathbf{w}_n^T \mathbf{x}_n > 1 \\ -y_n \mathbf{x}_n & \text{otherwise} \end{cases}$$

- SVM-SGD update rule:

$$\mathbf{w}_{n+1} = \mathbf{w}_n - \gamma \begin{cases} \lambda \mathbf{w}_n & \text{if } y_n \mathbf{w}_n^T \mathbf{x}_n > 1 \\ \lambda \mathbf{w}_n - y_n \mathbf{x}_n & \text{otherwise} \end{cases}$$



# Example – K-means

- sample loss function:

$$\hat{J}(\mathbf{w}_{1,n}, \dots, \mathbf{w}_{K,n}) = \min_{k=1 \dots K} \frac{1}{2} \|\mathbf{x}_n - \mathbf{w}_{k,n}\|^2$$

- Let  $k^* = \arg \min_{k=1 \dots K} \frac{1}{2} \|\mathbf{x}_n - \mathbf{w}_{k,n}\|^2$ , then

$$\hat{J}(\mathbf{w}_{1,n}, \dots, \mathbf{w}_{K,n}) = \frac{1}{2} \|\mathbf{x}_n - \mathbf{w}_{k^*,n}\|^2$$

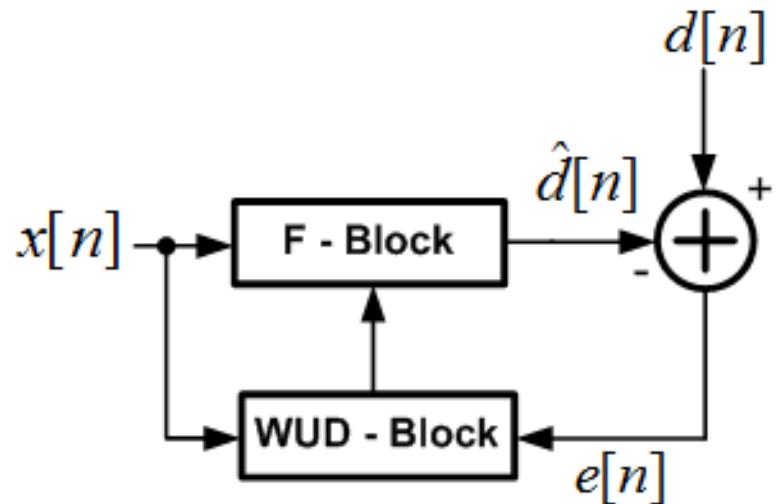
- sample gradient:

$$\nabla_{\mathbf{w}_{k,n}} \hat{J}(\mathbf{w}_{1,n}, \dots, \mathbf{w}_{K,n}) = - \begin{cases} \mathbf{x}_n - \mathbf{w}_{k,n} & \text{if } k = k^* \\ 0 & \text{otherwise} \end{cases}$$

- K-means-SGD update rule:

$$\begin{cases} \mathbf{w}_{k,n+1} = \mathbf{w}_{k,n} + \gamma(\mathbf{x}_n - \mathbf{w}_{k,n}) & \text{if } k = k^* \\ \mathbf{w}_{k,n+1} = \mathbf{w}_{k,n} & \text{otherwise} \end{cases}$$

# Example - LMS



$$\mathbf{w}_{n+1} = \mathbf{w}_n + \mu(-\hat{\nabla}_n)$$

- sample loss function:  $\hat{J}(\mathbf{w}) = e^2(n) = (d(n) - \mathbf{w}_n^T \mathbf{x}_n)^2$ ;

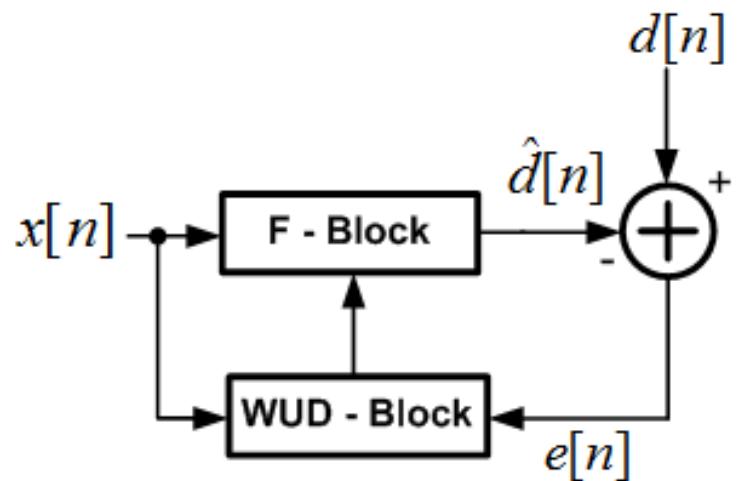
- sample gradient:

$$\begin{aligned}\hat{\nabla}_n &= \frac{\partial e^2(n)}{\partial \mathbf{w}_n} = 2e(n) \frac{\partial e(n)}{\partial \mathbf{w}_n} = 2e(n) \frac{\partial (d(n) - \mathbf{w}_n^T \mathbf{x}_n)}{\partial \mathbf{w}_n} \\ &= -2e(n)\mathbf{x}(n)\end{aligned}$$

- LMS update:

$$\mathbf{w}_{n+1} = \mathbf{w}_n + 2\mu e(n)\mathbf{x}(n)$$

# Example – Sign-LMS



$$\mathbf{w}_{n+1} = \mathbf{w}_n + \mu(-\hat{\nabla}_n)$$

- Loss function: absolute error
- The instantaneous estimate of the gradient is

$$\hat{\nabla}_n = \frac{\partial |e_n|}{\partial \mathbf{w}_n} = \text{sign}(e_n) \cdot \frac{\partial e_n}{\partial \mathbf{w}_n}$$

- Use chain rule + derivative of the absolute function is the sign function.
- But,
- So
- This gives the Sign-LMS equation:

$$\mathbf{w}_{n+1} = \mathbf{w}_n + \mu \cdot \text{sign}(e_n) \cdot \mathbf{x}_n$$

# Vanilla SGD

$$\theta = \theta - \eta \cdot \nabla_{\theta} J(\theta; x^{(i)}; y^{(i)})$$

LMS

$$e_n = y_n - \mathbf{w}_n^T \mathbf{x}_n \quad J(\mathbf{w}; \mathbf{x}_n; y_n) = e_n^2 = (y_n - \mathbf{w}_n^T \mathbf{x}_n)^2$$

$$\mathbf{w}_n = \mathbf{w}_n + \eta e_n \mathbf{x}_n \quad \nabla_{\mathbf{w}} J(\mathbf{w}) \Big|_{\mathbf{w}=\mathbf{w}_n} = -2e_n \mathbf{x}_n$$

# SGD Variants used in DNN Training

---

## An overview of gradient descent optimization algorithms\*

---

**Sebastian Ruder**  
Insight Centre for Data Analytics, NUI Galway  
Alyien Ltd., Dublin  
[ruder.sebastian@gmail.com](mailto:ruder.sebastian@gmail.com)

(arxiv, June 2017)

# Momentum

$$\begin{array}{c} g_t \\ \downarrow \\ v_t = \gamma v_{t-1} + \eta \nabla_{\theta} J(\theta) \\ \theta = \theta - v_t \\ t+1 \qquad t \end{array} \rightarrow \text{gradient averaging} \rightarrow v_t = \eta \sum_{i=0}^t g_{t-i}$$

- $\gamma \sim 0.9; \eta \sim 0.01$
- employs gradient/update history to generate current update
- dampens oscillations near local minima

# Nestorov Acceleration

$$v_t = \gamma v_{t-1} + \eta \nabla_{\theta} J(\theta_t - \gamma v_{t-1})$$
$$\theta_{t+1} \approx \theta_t - v_t$$

$\approx g_{t+1}$



- $\gamma \sim 0.9; \eta \sim 0.01$
- similar to momentum except use predicted/future gradient to make update

# AdaGrad

[Duchi et al., 2011]

$$g_{t,i} = \nabla_{\theta_t} J(\theta_{t,i}) \quad \rightarrow \quad \theta_{t+1,i} = \theta_{t,i} - \eta \cdot g_{t,i} \quad \text{Vanilla SGD}$$

$$G_{t,ii} = \sum_{k=1}^t g_{k,i}^2 \quad \text{accumulate squared gradients from 1 to } t$$

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i} \quad \text{AdaGrad update}$$

- **adapts** learning rates **per dimension** and over time
- works well with sparse gradients

# RMSProp

$$E[g^2]_t = 0.9E[g^2]_{t-1} + 0.1g_t^2 \rightarrow \text{exponentially decaying average squared gradient}$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

- includes a forgetting factor
- works well on-line and non-stationary settings

# ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION

Diederik P. Kingma\*  
University of Amsterdam, OpenAI  
[dpkingma@openai.com](mailto:dpkingma@openai.com)

Jimmy Lei Ba\*  
University of Toronto  
[jimmy@psi.utoronto.ca](mailto:jimmy@psi.utoronto.ca)

- ADAM - adaptive momentum estimation
- computes 1<sup>st</sup> and 2<sup>nd</sup> moments of gradient
- combines advantages of AdaGrad and RMSProp
- Advantages:
  - updates are invariant to scaling of gradient
  - step-size is bounded by step-size hyperparameter
  - naturally performs step-size annealing
  - works with sparse gradients

# First & Second Moments

- gradient at sample index  $t$ :  $g_t$
- first moment:  $m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$
- second moment:  $\nu_t = \beta_2 \nu_{t-1} + (1 - \beta_2) g_t^2$
- bias correction: to make exponentially decaying averages approach true average, i.e.,  $m_t \rightarrow \mathbb{E}[g_t]$
- bias-corrected 1<sup>st</sup> moment:  $\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$
- bias-corrected 2<sup>st</sup> moment:  $\hat{\nu}_t = \frac{\nu_t}{1 - \beta_2^t}$

# Bias Correction – for 2<sup>nd</sup> Moment

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \quad \xrightarrow{\text{loop unroll}} \quad v_t = (1 - \beta_2) \sum_{i=1}^t \beta_2^{t-i} \cdot g_i^2$$

$$\begin{aligned}\mathbb{E}[v_t] &= \mathbb{E} \left[ (1 - \beta_2) \sum_{i=1}^t \beta_2^{t-i} \cdot g_i^2 \right] \\ &= \mathbb{E}[g_t^2] \cdot (1 - \beta_2) \sum_{i=1}^t \beta_2^{t-i} + \zeta \\ &= \mathbb{E}[g_t^2] \cdot (1 - \beta_2^t) + \zeta\end{aligned}$$

- for stationary  $g_t \rightarrow \zeta = 0$

# Adam Update Rule

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

---

**Algorithm 1:** *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation.  $g_t^2$  indicates the elementwise square  $g_t \odot g_t$ . Good default settings for the tested machine learning problems are  $\alpha = 0.001$ ,  $\beta_1 = 0.9$ ,  $\beta_2 = 0.999$  and  $\epsilon = 10^{-8}$ . All operations on vectors are element-wise. With  $\beta_1^t$  and  $\beta_2^t$  we denote  $\beta_1$  and  $\beta_2$  to the power  $t$ .

---

**Require:**  $\alpha$ : Stepsize

**Require:**  $\beta_1, \beta_2 \in [0, 1]$ : Exponential decay rates for the moment estimates

**Require:**  $f(\theta)$ : Stochastic objective function with parameters  $\theta$

**Require:**  $\theta_0$ : Initial parameter vector

$m_0 \leftarrow 0$  (Initialize 1<sup>st</sup> moment vector)

$v_0 \leftarrow 0$  (Initialize 2<sup>nd</sup> moment vector)

$t \leftarrow 0$  (Initialize timestep)

**while**  $\theta_t$  not converged **do**

$t \leftarrow t + 1$

$g_t \leftarrow \nabla_{\theta} f_t(\theta_{t-1})$  (Get gradients w.r.t. stochastic objective at timestep  $t$ )

$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$  (Update biased first moment estimate)

$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$  (Update biased second raw moment estimate)

$\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$  (Compute bias-corrected first moment estimate)

$\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$  (Compute bias-corrected second raw moment estimate)

$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$  (Update parameters)

**end while**

**return**  $\theta_t$  (Resulting parameters)

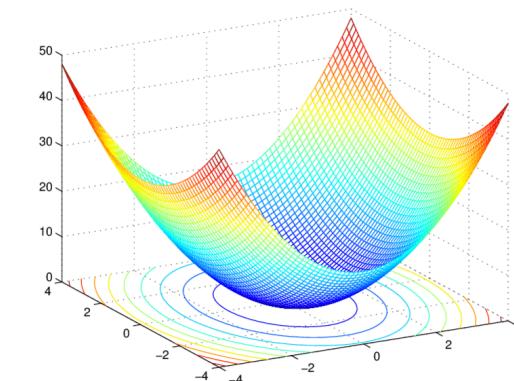
---

# SGD Tricks & Tweaks

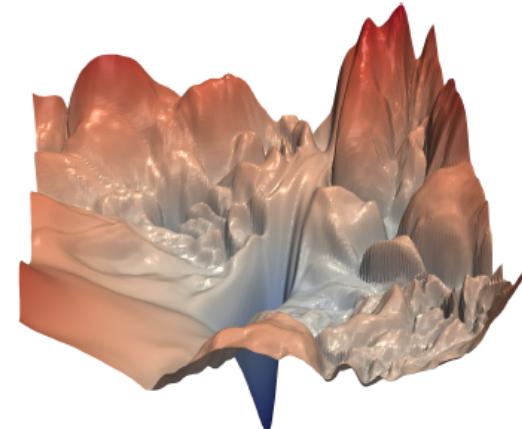
# Initialization

- random initializing of linear regressor works → regressor converges to optimum regardless of initial conditions (convex loss function)

$$\mathbf{w}_n = \mathbf{w}_n + \eta e_n \mathbf{x}_n$$



- DNN loss function is non-convex → need to initialize properly: what does this mean?



# Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification

Kaiming He

Xiangyu Zhang

Shaoqing Ren

Jian Sun

Microsoft Research

{kahe, v-xiangz, v-shren, jiansun}@microsoft.com

- account for: 1) presence of ReLU, and 2) cascaded architecture

$$\mathbf{y}_l = \mathbf{W}_l \mathbf{x}_l + \mathbf{b}_l$$

$$n_l = \dim(\mathbf{x}_l)$$

L-layer network

sufficient condition for convergence to a scalar

$$Var[y_L] = Var[y_1] \left( \prod_{l=2}^L \frac{1}{2} n_l Var[w_l] \right) \rightarrow$$

$$\frac{1}{2} n_l Var[w_l] = 1$$

- initialize weights as  $w_l \sim N(0, \sigma_w^2)$  with  $\sigma_w^2 = \frac{2}{n_l}$
- similar initialization in backward path with proper  $n_l$

# Batch Norm

Batch Normalization: Accelerating Deep Network Training by  
Reducing Internal Covariate Shift

Arxiv, 2015

Sergey Ioffe

Google Inc., [sioffe@google.com](mailto:sioffe@google.com)

Christian Szegedy

Google Inc., [szegedy@google.com](mailto:szegedy@google.com)

**Input:** Values of  $x$  over a mini-batch:  $\mathcal{B} = \{x_1 \dots m\}$ ;

Parameters to be learned:  $\gamma, \beta$

**Output:**  $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i \quad // \text{mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 \quad // \text{mini-batch variance}$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad // \text{normalize}$$

$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i) \quad // \text{scale and shift}$$

- ideally want to whiten (decorrelate the input) into a layer (why?) Because it speeds-up convergence
- Too expensive → standardize inputs into each layer:

$$\hat{x}^{(k)} = \frac{x^{(k)} - \mathbb{E}[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}$$

- what is  $\mathbb{E}[\hat{x}^{(k)}]$  and  $\text{Var}(\hat{x}^{(k)})$ ?
- But, need to ensure the network's representational power is preserved:

$$y^{(k)} = \gamma^{(k)} \hat{x}^{(k)} + \beta^{(k)}$$

- learn the  $\beta$ s and  $\gamma$ s during training

# Learning Schedule

- variable step-size (gear shifting) – tries to achieve fast convergence and high accuracy simultaneously
- Learning rate schedule: reduces  $\mu$  as convergence proceeds
  - large initial  $\mu$  speeds up convergence
  - small later  $\mu$  later results in higher accuracy
- E.g., reduce  $\mu$  by factor of 2 until precision limits are reached

# Drop-out

Journal of Machine Learning Research 15 (2014) 1929-1958

Submitted 11/13; Published 6/14

## Dropout: A Simple Way to Prevent Neural Networks from Overfitting

Nitish Srivastava

Geoffrey Hinton

Alex Krizhevsky

Ilya Sutskever

Ruslan Salakhutdinov

NITISH@CS.TORONTO.EDU

HINTON@CS.TORONTO.EDU

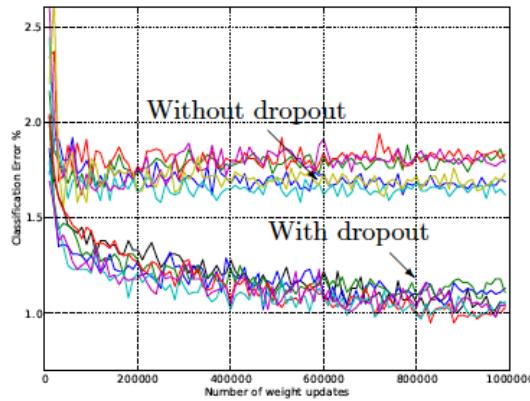
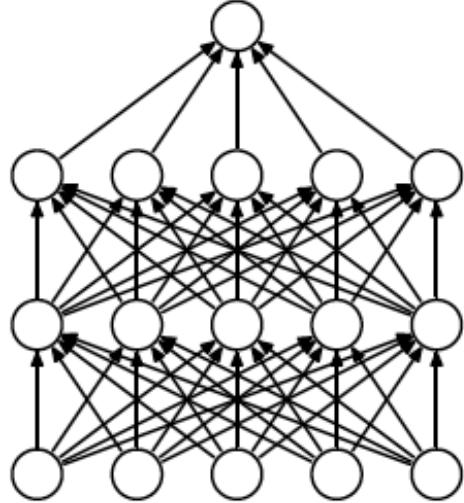
KRIZ@CS.TORONTO.EDU

ILYA@CS.TORONTO.EDU

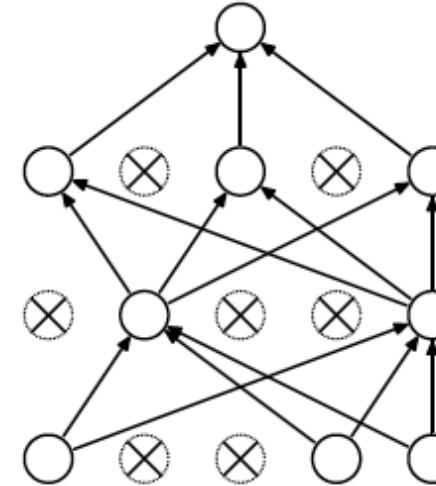
RSALAKHU@CS.TORONTO.EDU

- a regularization technique
- ideal regularizer: average of predictions of an ensemble of networks (model combining)
- hard to realize → need multiple networks architectures or train with different data
- dropout is an efficient way to do this

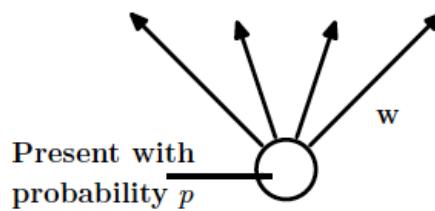
**standard network**



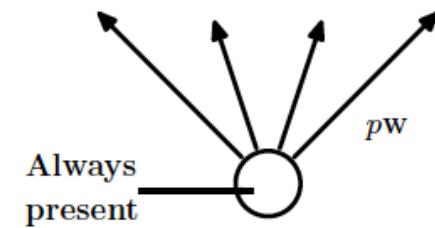
**dropout thinned network**



**during training**

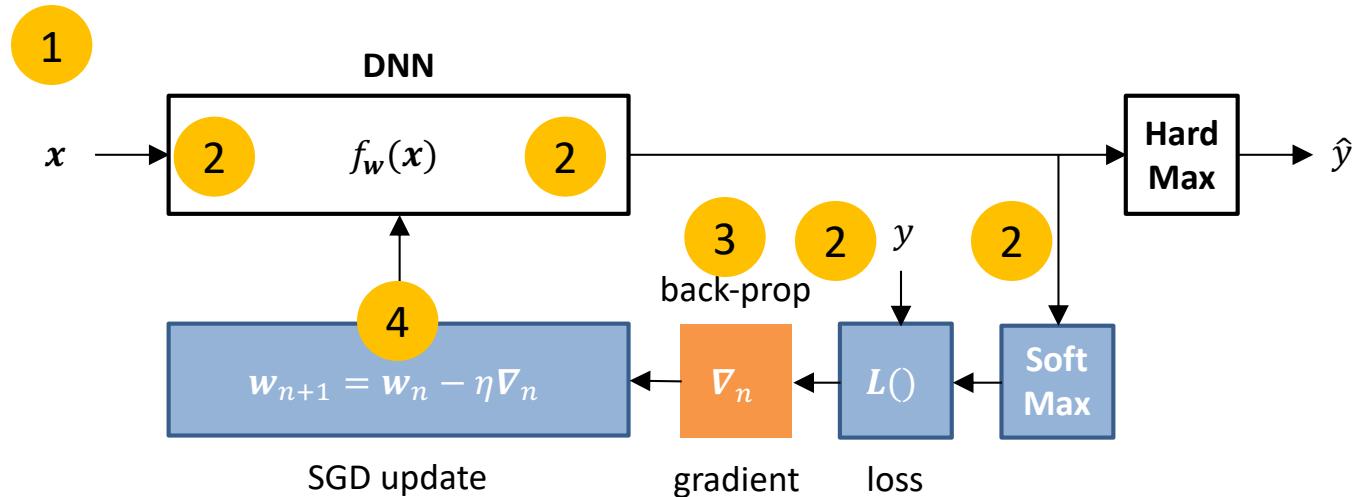


**during test**



- retain each node with probability  $p$  per sample
- $p$ : retain probability of a unit/node  $\sim 0.5$  to  $1.0$
- trains an ensemble of  $2^N$  networks ( $N$ : number of nodes)

# Summary- Key Steps in Training a DNN



- **Step 1:** fetch one batch of inputs
- **Step 2:** forward propagation and comparison of output with true label (loss function)
- **Step 3:** back-propagation of gradients with respect to loss function
- **Step 4:** weight update  $w \leftarrow w - \mu \nabla_w$  (or other update rule)
- **Steps 1-4** are repeated until satisfactory convergence

# Summary

- Trade-off between complexity and accuracy
- Based on design intuitions to preserve accuracy
- separable convolutions
- use of point-wise convolutions
- delaying max-pooling
- others

## Course Web Page

<https://courses.grainger.illinois.edu/ece598nsg/fa2020/>

<https://courses.grainger.illinois.edu/ece498nsu/fa2020/>

<http://shanbhag.ece.uiuc.edu>