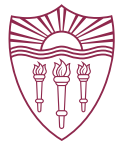
USC Viterbi
School of Engineering



INTRODUCTION AND REVIEW

EE 641 – UNIT 1

DR. BRANDON FRANZKE

Fall 2025

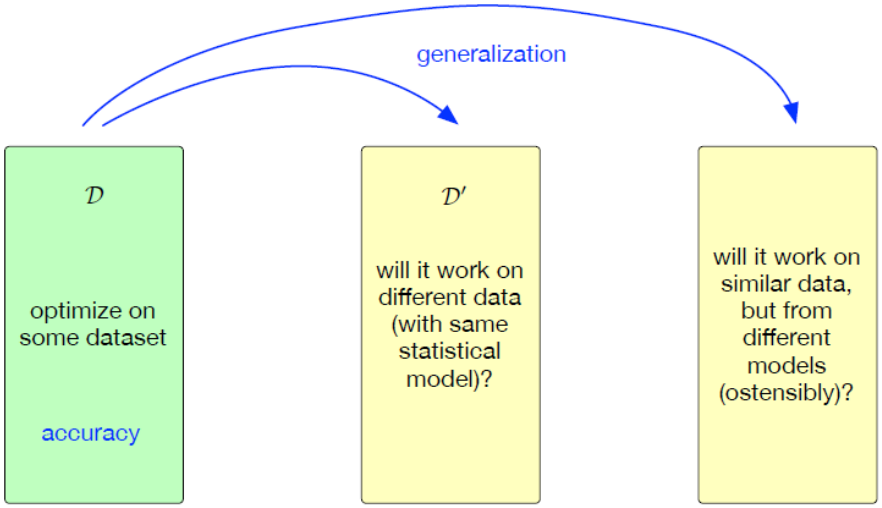
OUTLINE FOR SLIDES

- Machine Learning goals and limitations
- Estimation: regression and maximum likelihood
- Fully connected multilayer perceptron (MLP) networks
- Backpropagation
- Dealing with Data
- PyTorch and training
- Convolutional Neural Networks

EE 641 TOPICS

- Specialized CNN Architectures
  - Segmentation
- Generative Adversarial Networks (GANS)
- Recurrent Neural Networks (RNN)
- Reinforcement Learning (RL)
- Transformer architectures
- Hyperparameter tuning and AutoML
- Diffusion and Particle Models
- MLOps and Machine Learning pipelines

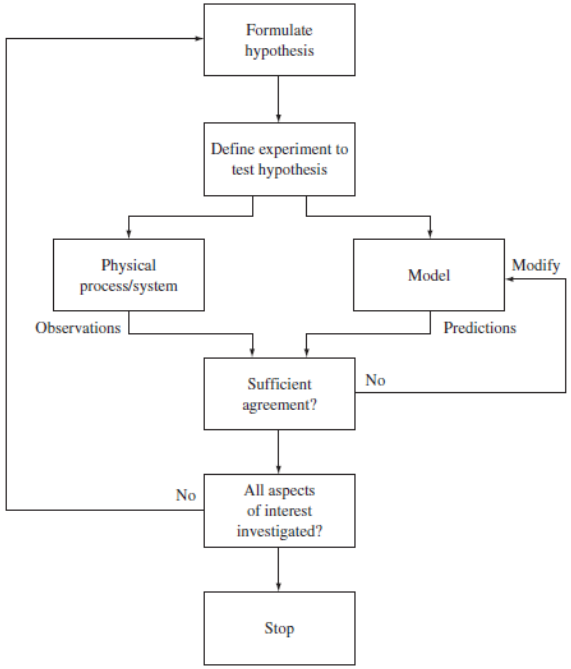
GENERALIZATION IS THE GOAL OF MACHINE LEARNING



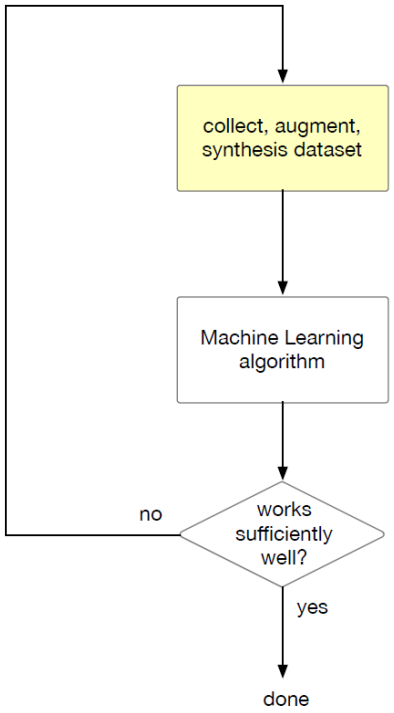
- Do **not** care about the performance on the dataset we have
- **Do care about performance on similar data that has no labels**
- **Accuracy/Generalization trade-off (aka bias-variance trade):**
  - **Generally: optimize accuracy to the extreme reduces capability to generalize**

MACHINE LEARNING ⟹ "*DATA IS* THE STORY"

EXPERIMENT DRIVEN MODELING
(CLASSICAL)



Data driven modeling



A. Leon-Garcia. Probability, Statistics, and Random Processes for EE. 2008.

All models are wrong, but some are useful…

2.3 Parsimony

Since all models are wrong the scientist cannot obtain a "correct" one by excessive elaboration. On the contrary, following William of Occam he should seek an economical description of natural phenomena. Just as the ability to devise simple but evocative models is the signature of the great scientist so overelaboration and overparameterization is often the mark of mediocrity.

2.4 Worrying Selectively

Since all models are wrong the scientist must be alert to what is importantly wrong. It is inappropriate to be concerned about mice when there are tigers abroad.

George Box

"Science and statistics" Journal of the American Statistical Association. 1976.

CONDA ENVIRONMENT

```
conda create --name ee641_work python=3.11
conda activate ee641_work


# mac (mps)
#conda install pytorch::pytorch torchvision torchaudio -c pytorch
# pc (with cuda), check cuda version e.g. > 11.7 / 11.8
#conda install pytorch torchvision torchaudio pytorch-cuda -c pytorch -c
nvidia


conda install numpy scipy opencv matplotlib
conda install h5py jupyter tensorboard seaborn tqdm pandas
conda install -c conda-forge torchinfo
# extra
pip install torchviz
```

ESTIMATION AND REGRESSION

PROBLEM: ESTIMATE $\mathbf{y}(t)$ FROM $\boldsymbol{X}(t) = \boldsymbol{x}$

- **Problem**: Given a vector observation $\boldsymbol{X}(t) = \boldsymbol{x}$, we would like to estimate $y(t)$ via linear filter $\hat{y} = \boldsymbol{w}^T \boldsymbol{x}$ with minimized mean squared error (MSE).

- The objective (cost function) to be minimized is $\mathrm{MSE} = \mathbb{E}\{[y(t) - \boldsymbol{w}^T \boldsymbol{x}]^2\}$. The filter design variables are $\boldsymbol{w}$.

LMMSE ASSUMPTIONS

We know the second order statistics of the joint distribution $p_{\boldsymbol{X}(t),Y(t)}(\boldsymbol{x}, y)$

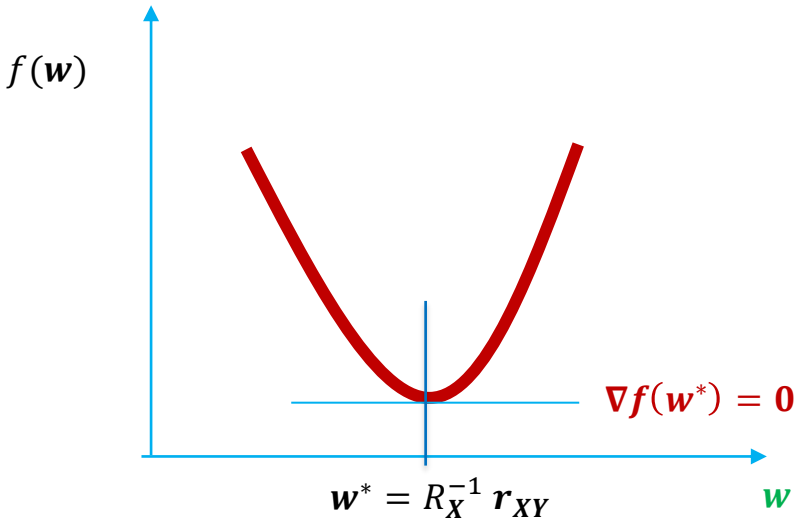
$$i.e.,\ \boldsymbol{m_X}, m_Y, R_{\boldsymbol{X}}, r_Y, \boldsymbol{r}_{XY}.$$

The objective is quadratic in $\boldsymbol{w}$,

$$f(\boldsymbol{w}) = \mathbb{E}\{[Y(t) - \boldsymbol{w}^T\boldsymbol{x}]^2\}$$
$$= r_Y - 2\boldsymbol{r}_{Y\boldsymbol{X}}^T\boldsymbol{w} + \boldsymbol{w}^T R_{\boldsymbol{X}}\boldsymbol{w}$$

The global optimal occurs at $\boldsymbol{w}^*$.

$$i.e.,\ \nabla f(\boldsymbol{w})|_{\boldsymbol{w}=\boldsymbol{w}^*} = \boldsymbol{0}$$



MMSE ESTIMATION: SPECIAL CASE JOINTLY-GAUSSIAN

Estimate $\mathbf{Y(t)}$ from $\mathbf{X(t) = x}$

Conditional Gaussian pdf:

$$p_{y(t)|x(t)}(y|x) = \frac{p_{X(t),Y(t)}(x,y)}{p_{X(t)}(x)}$$

$$= \frac{\mathcal{N}_{M+N}\left(\begin{bmatrix}x\\y\end{bmatrix}; \begin{bmatrix}m_X\\m_Y\end{bmatrix}, \begin{bmatrix}K_Y & K_{XY}\\K_{YX} & K_Y\end{bmatrix}\right)}{\mathcal{N}_N(x; m_X, K_X)}$$

$$= \mathcal{N}_M\left(y; \underbrace{m_Y + K_{YX}K_X^{-1}(x - m_X)}_{\text{AMMSE estimator}}, \underbrace{K_Y - K_{YX}K_X^{-1}K_{XY}}_{\text{error covariance}}\right)$$

JG… no ML or deep learning needed!

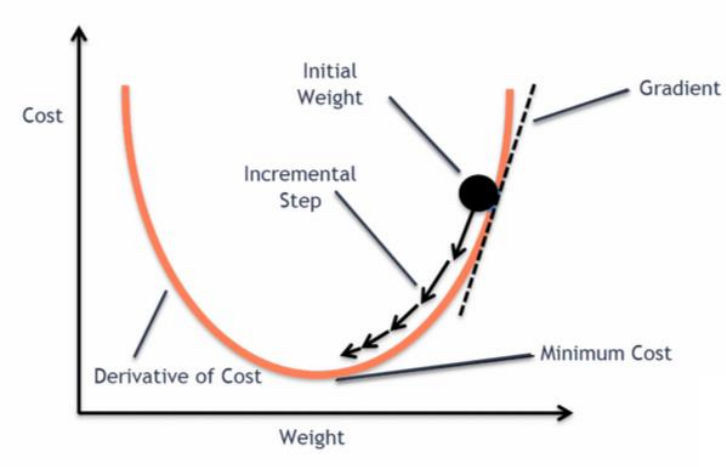
For **JG** observation and target: $\mathbb{E}[Y|x]$ **is the Affine MMSE estimator**

GRADIENT DESCENT

- Instead of obtaining optimal $\boldsymbol{w}^*$ directly, we can also find it iteratively via

1. Initialize $\boldsymbol{w}_0$
2. $\boldsymbol{w}_{n+1} = \boldsymbol{w}_n - \eta \, \nabla f(\boldsymbol{w}_n)$

$= \boldsymbol{w}_n + \eta(\boldsymbol{r}_{xy} - R_x \boldsymbol{w}_n)$



source: https://blog.clairvoyantsoft.com/

REMARKS

1. **Closed-form global optimality** can be derived if we have a convex and differentiable cost function.

2. **Gradient descent** works in any differentiable cost function.

3. What if we don't have second order statistics but lots of samples? Use LMS!

   Single point gradient descent or small batch gradient descent!

R_x computation not tractable for large number of samples

STEEPEST DESCENT AND LMS

Estimate y(u) from x(u) = x

$$E(w) = \mathbb{E}\{[y(t) - w^T x(t)]^2\}$$

$$\nabla_w E = 2\mathbb{E}\{wx(t)x^T(t) - y(t)x(t)\}$$

$$= 2\left(w^t R_x - r_{xy}\right)$$

Steepest descent using (ensemble average) gradient:

$$\hat{w}_{n+1} = \hat{w}_n - \left(\frac{\eta}{2}\right)\nabla_w E$$

$$= \hat{w}_n + \eta\left(r_{xy} - R_x \hat{w}_n\right)$$

Single Point Stochastic Gradient Descent:

$$-\frac{1}{2}\nabla_w E = r_{xy} - R_X w$$

$$= \mathbb{E}\{y(t)X(t) - X(t)X^T(t)w\}$$

$$\approx y_n x_n - x_n x_n^T w$$

$$= (y_n - x_n^T \hat{w}_n)x_n$$

$$\boxed{\hat{w}_{n+1} = \hat{w}_n + \eta(y_n - \hat{w}_n^T x_n)x_n}$$

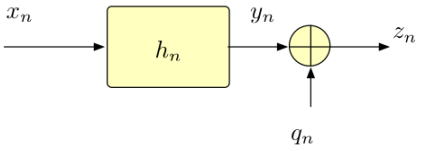
this is called "**on-line learning**"

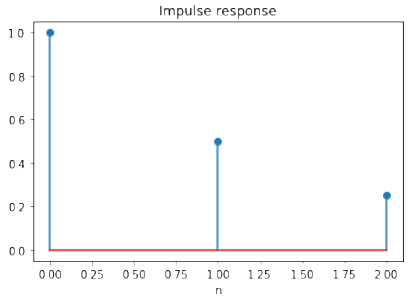
when $n \sim$ time, this is the **Adaptive** Least Mean Square (LMS) filter

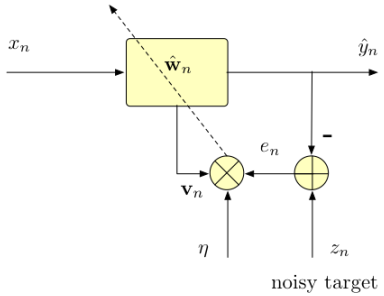
when n does not represent time can average the gradient over more data points to improve approximation (batching)

EXPERIMENT: LMS EXAMPLE

Generate data:

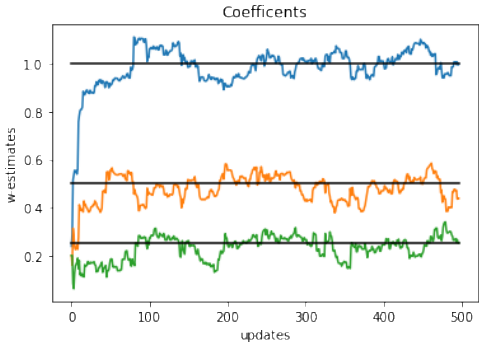




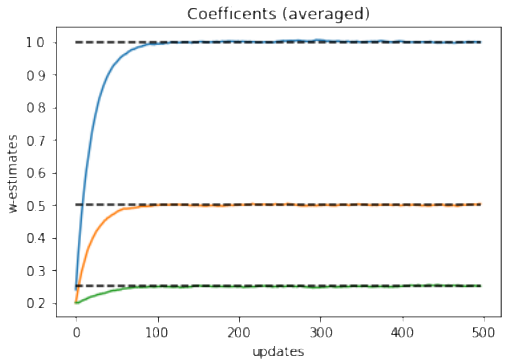


this is the ideal case where model and observed data are matched

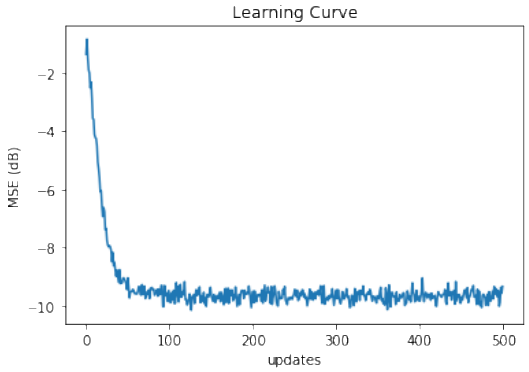
$\eta = 0.05$, SNR = 10 dB



single run



averaged over 500 runs



averaged over 500 runs

GENERAL REGRESSION PROBLEM

Given a data set: $\mathcal{D} = \{(\mathbf{x}_n, \mathbf{y}_n)\}_{n=1}^{N}$

General regression problem:

$$\min_{\Theta} \langle C(\mathbf{y}, \mathbf{g}(\mathbf{x}; \Theta)) \rangle_{\mathcal{D}} \qquad \Theta_{opt} = \arg\min_{\Theta} \langle C(\mathbf{y}, \mathbf{g}(\mathbf{x}; \Theta)) \rangle_{\mathcal{D}} \qquad \hat{\mathbf{y}} = \mathbf{g}(\mathbf{x}; \Theta_{\text{opt}})$$

Empirical expectation (average over data):

$$\langle \mathbf{h}(\mathbf{x}, \mathbf{y}) \rangle_{\mathcal{S}} \equiv \frac{1}{|\mathcal{S}|} \sum_{(\mathbf{x}_n, \mathbf{y}_n) \in \mathcal{S}} \mathbf{h}(\mathbf{x}_n, \mathbf{y}_n)$$

x ~ *regressor* (*observed*)
y ~ target (*desired*)

For large averaging sets (*i.e.*, many realizations):

$$\mathbb{E}\{\mathbf{h}(\mathbf{x}(t), \mathbf{y}(t))\} = \int \mathbf{h}(\mathbf{x}, \mathbf{y}) p_{x(t),y(t)}(\mathbf{x}, \mathbf{y})\, d\mathbf{y}\, d\mathbf{x} \approx \langle \mathbf{h}(\mathbf{x}, \mathbf{y}) \rangle_{\mathcal{S}}$$

sample mean

Monte Carlo method (see: Markov Chain Monte Carlo, **MCMC**)

LEAST-SQUARES (LS) REGRESSION PROBLEM

$$\min_{\Theta} \langle \|\mathbf{y} - \mathbf{g}(\mathbf{x}; \Theta)\| \rangle_{\mathcal{D}} \quad \Leftrightarrow \quad \min_{\Theta} \sum_{n=1}^{N} \|\mathbf{y}_n - \mathbf{g}(\mathbf{x}_n; \Theta)\|^2$$

$$\Theta_{\text{opt}} = \arg\min_{\Theta} \langle \|\mathbf{y} - \mathbf{g}(\mathbf{x}; \Theta)\|^2 \rangle_{\mathcal{D}}$$

Squared-error is a common cost function in (electrical) engineering

corresponds to **power or energy** in many applications
(Parseval's Theorem)

LINEAR AND AFFINE REGRESSION SOLUTION

Data averaging operator has linearity property like expectation

$$\mathbb{E}\{\mathrm{L}(x(t))\} = \mathrm{L}\left(\mathbb{E}(x(t))\right) \qquad \langle \mathrm{L}(x)\rangle = \mathrm{L}(\langle x\rangle)$$

This means the solutions are the same as the MMSE solutions with expectation replaced by data average

For example, Linear LS regression:

$$\mathbf{W}_{\mathrm{LLSE}} = \widehat{\mathbf{R}}_{\mathbf{YX}}\widehat{\mathbf{R}}_{\mathbf{X}}^{-1}$$

$$\hat{\mathbf{y}} = \widehat{\mathbf{R}}_{\mathbf{YX}}\widehat{\mathbf{R}}_{\mathbf{X}}^{-1}\mathbf{x}$$

$$\begin{aligned}\varepsilon_{LLS} &= \langle \|\boldsymbol{y} - \mathbf{W}_{\mathrm{LLSE}}\mathbf{x}\|^2\rangle \\ &= \langle \|\mathbf{y}\|^2 - \|\mathbf{W}_{\mathrm{LLSE}}\mathbf{x}\|^2\rangle_{\mathcal{D}} \\ &= \mathrm{Tr}\left(\widehat{\mathbf{R}}_{\mathbf{Y}} - \widehat{\mathbf{R}}_{\mathbf{YX}}\widehat{\mathbf{R}}_{\mathbf{X}}^{-1}\,\widehat{\mathbf{R}}_{\mathbf{XY}}\right)\end{aligned}$$

$$\widehat{\mathbf{R}}_{\mathbf{X}} = \langle \mathbf{x}\mathbf{x}^T\rangle_{\mathcal{D}} = \frac{1}{N}\sum_{n=1}^{N}\mathbf{x}_n\mathbf{x}_n^T$$

$$\widehat{\mathbf{R}}_{\mathbf{XY}} = \langle \mathbf{x}\mathbf{y}^T\rangle_{\mathcal{D}} = \frac{1}{N}\sum_{n=1}^{N}\mathbf{x}_n\mathbf{y}_n^T$$

LINEAR CLASSIFIER

perform linear regression and then **threshold** to hard decision

Example:

$$y \in \{-1, +1\}$$

$$\hat{y} = sign(\mathbf{w}^T \mathbf{x})$$

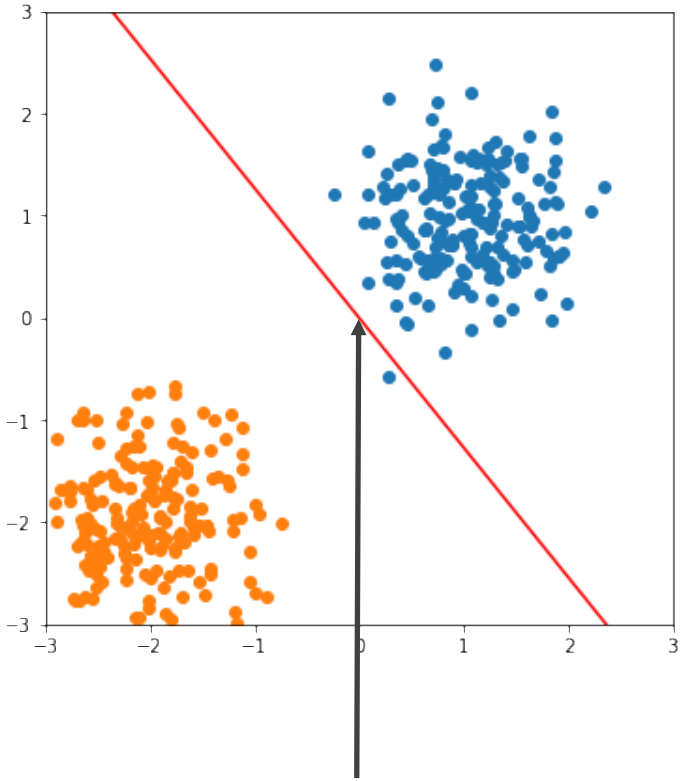
$$sign(v) = \begin{cases} +1, & v \geq 0 \\ -1, & v < 0 \end{cases}$$

$$\min_{\mathbf{w}} \frac{1}{N} \sum_{n=1}^{N} (y_n - \mathbf{w}^T \mathbf{x}_n)^2$$

standard LLSE regression with prediction thresholding

EXAMPLE: LINEAR AND AFFINE REGRESSION

$$\hat{y} = \mathbf{x}^T\mathbf{w}$$

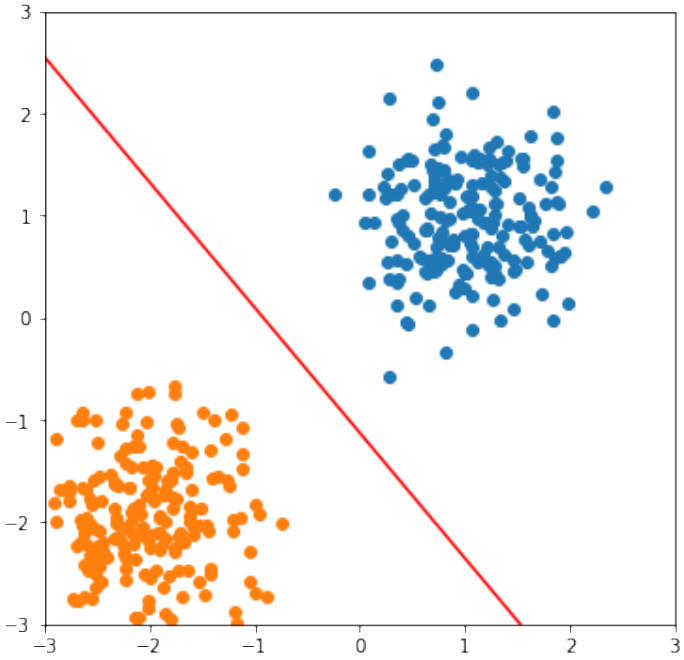


for the case with no bias term, the decision threshold has to pass through the origin

$$s_0 = \begin{bmatrix} +1 \\ +1 \end{bmatrix}$$

$$s_1 = \begin{bmatrix} -2 \\ -2 \end{bmatrix}$$

$$\hat{y} = [\mathbf{x}^T \quad | \quad 1] \begin{bmatrix} \mathbf{w} \\ b \end{bmatrix}$$



adding the bias term allows for offset from the origin

MAXIMUM LIKELIHOOD ESTIMATION EXAMPLE

this is a model for the data $\{(x_n, y_n)\}$:

$$y_n = \mathbf{w}^T \mathbf{x}_n + v_n, \qquad n = 1,2,\dots,N$$

$$\mathbf{y} = \mathbf{X}\mathbf{w} + \mathbf{v}(t)$$

$$p_{v(t)}(\mathbf{v}) = \mathcal{N}_N(\mathbf{v}; \mathbf{0}, \sigma_v^2 \mathbf{I})$$

$$p_{\mathbf{y}(t)|\mathbf{X}(t)}(\mathbf{y}|\mathbf{X}; \mathbf{w}) = p_{\mathbf{v}(t)}(\mathbf{y} - \mathbf{X}\mathbf{w}) = \mathcal{N}_N(\mathbf{v}; \mathbf{X}\mathbf{w}, \sigma_v^2 \mathbf{I})$$

$$NLL(\mathbf{w}) = -\ln\left(p_{\mathbf{y}(t)}(\mathbf{y}|\mathbf{X}; \mathbf{w})\right)$$

$$= -\ln\left(\frac{1}{(2\pi\sigma_v^2)^{\frac{N}{2}}} \exp\left[-\frac{1}{2\sigma_v^2}\|\mathbf{y} - \mathbf{X}\mathbf{w}\|^2\right]\right)$$

$$= -\frac{1}{2\sigma_v^2}\|\mathbf{y} - \mathbf{X}\mathbf{w}\|^2 + \frac{\mathrm{N}}{2}\ln(2\pi\sigma_v^2)$$

$$\max_{\mathbf{w}} p_{\mathbf{y}(t)|\mathbf{X}(t)}(\mathbf{y}|\mathbf{X}; \mathbf{w}) \Leftrightarrow \min_{\mathbf{w}} \|\mathbf{y} - \mathbf{X}\mathbf{w}\|^2$$

Maximum Likelihood <==> Minimize Neg-Log-Likelihood <==> LLSE regression

(under **this** model for the data)

ML ESTIMATION AND INFORMATION THEORY

Entropy:

$$H(X(t)) = \mathbb{E}\left\{\log_2\left(\frac{1}{p_{X(t)}(X(t))}\right)\right\}$$
$$= \sum_k p_{X(t)}(k)\log_2\left(\frac{1}{p_{X(t)}(k)}\right)$$
$$= \sum_k p_k \log_2\left(\frac{1}{p_k}\right)$$

Intuition:

events with low probability have large information — *e.g.*, "it will snow in Phoenix tomorrow"

the entropy is the average information learned when the value of $X(u)$ is revealed.

Examples:

weather report in Phoenix has low entropy (almost always the same), whereas in Sioux City, SD it has high entropy (highly variant weather)

fair die:

$$H(X(t)) = \log_2(1/6) = 2.58\ bits/roll$$

loaded die:

$$H(X(t)) = -0.4\log_2(0.4) - 0.1\log_2(0.1) - 0.01\log_2(0.01)$$
$$- 0.09\log_2(0.09) - 0.25\log_2(0.25) - 0.15\log_2(0.15)$$
$$= 2.15\ bits/roll$$

ML ESTIMATION AND INFORMATION THEORY

KL-Divergence

$$D(p \parallel \tilde{p}) = \mathbb{E}_p\left\{\log\left(\frac{p_x(X(t))}{\tilde{p}_x(X(t))}\right)\right\}$$

$$= \sum_k p_k \log\left(\frac{p_k}{\tilde{p}_k}\right)$$

$$= \sum_k p_k \log(p_k) - \sum_k p_k \log(\tilde{p}_k)$$

$$= CE(p, \tilde{p}) - H(p)$$

Cross-Entropy

$$CE(p, \tilde{p}) = \mathbb{E}_p\left\{\log\left(\frac{1}{\tilde{p}(X(t))}\right)\right\}$$

MULTI-CLASS CROSS ENTROPY EXAMPLE

One hot encoding:

cat: 0
dog: 1
bird: 2

Sample data labels:

n=1: y=1 (dog)
n=2: y=2 (bird)
n=3: y=0 (cat)

Classifier Output:

n=1: [0.3, 0.5, 0.2]
n=2: [0, 0, 1]
n=3: [0.4, 0.5, 0.1]

[p(cat), p(dog), p(bird)]

$$Loss = -\frac{1}{3}[\log(0.5) + \log(1) + \log(0.4)]$$

$$\overline{MCE}(p_{data}, p_{model}(w)) = -\frac{1}{N}\sum_{n=1}^{N}\sum_{m=1}^{M} \mathbb{I}[y_n = m] \log(p_{model}(y_n = m; w))$$

LOGISTIC REGRESSION MOTIVATION

This motivates a model for a binary variable y:

$$\tilde{y}(u) \sim Bernoulli(p)$$

$$p = p_1 = \sigma(\boldsymbol{w}^T\boldsymbol{x}) = \frac{1}{1+\exp(-\boldsymbol{w}^T\boldsymbol{x})}$$

$$(1-p) = p_0 = \frac{1}{1+\exp(+\boldsymbol{w}^T\boldsymbol{x})}$$

Model the values of the binary targets

$$\tilde{y}_n \sim Bernoulli\big(\sigma(\boldsymbol{w}^T\boldsymbol{x})\big) \qquad \text{i.i.d.}$$

LOGISTIC REGRESSION

Take ML approach to determining $\boldsymbol{w}$ for this model:

$$p(y|X;w) = \prod_{n=1}^{N} p_n^{\tilde{y}_n}[1-p_n]^{(1-\tilde{y}_n)} \qquad p_n = \sigma(\boldsymbol{w}^T \boldsymbol{x}_n)$$

$$= \prod_{n=1}^{N} p_n^{\mathbb{I}[y_n=+1]}[1-p_n]^{\mathbb{I}[y_n=-1]}$$

$$p_n^{\tilde{y}_n}[1-p_n]^{(1-\tilde{y}_n)} = \begin{cases} p_n & \tilde{y}_n = 1 \ \ (y_n = +1) \\ 1-p_n & \tilde{y}_n = 0 \ \ (y_n = -1) \end{cases}$$

The negative log-likelihood is….

Example:

Labels ($\tilde{y}$): 1, 0, 1

Output ($\sigma(w^T x)$): 0.9, 0.1, 0.2

$p(y|x;w)$: 0.9 * 0.9 * 0.2

LOGISTIC REGRESSION

The negative log-likelihood is:

$$NLL(\boldsymbol{w}) = -\left(\sum_{n=1}^{N} \tilde{y}_n \log(p_n) + (1-\tilde{y}_n)\log(1-p_n)\right) \qquad p_n = \sigma(\boldsymbol{w}^T \boldsymbol{x}_n)$$

$$= -\left(\sum_{n=1}^{N} \tilde{y}_n \log\big(\sigma(\boldsymbol{w}^T \boldsymbol{x})\big) + (1-\tilde{y}_n)\log\big(1-\sigma(\boldsymbol{w}^T \boldsymbol{x})\big)\right)$$

$$= \sum_{n=1}^{N} \log(1+\exp[-y_n \boldsymbol{w}^T \boldsymbol{x}])$$

This is called the binary cross-entropy loss. We will see that this can be viewed as a distance between two distributions

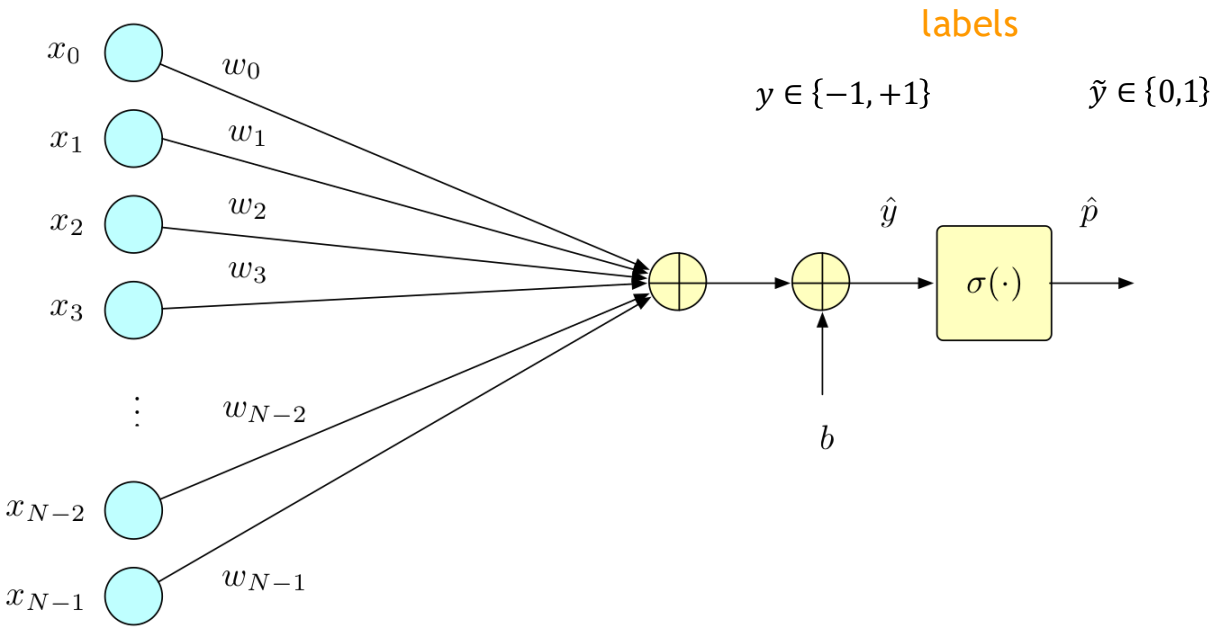
Example:

Labels ($\tilde{y}$): 1, 0, 1

Output ($\sigma(w^T x)$): 0.9, 0.1, 0.2

$NLL(w)$: 0.11 + 0.11 + 1.6

LOGISTIC REGRESSION



Two problems addressed:

1. Logistic function maps LLR to p_1
2. Binary cross-entropy is the loss that arises from a Bernoulli model and maximum likelihood parameter estimation.

LOGISTIC REGRESSION

Summary:

Logistical regression is ML estimation of $\boldsymbol{w}$ for an i.i.d. Bernoulli model with

$$p_n = \sigma(\boldsymbol{w}^T \boldsymbol{x})$$

which can be viewed as regression with the (empirical) binary cross-entropy cost function

no closed form, usually use SGD to perform the regression

We will see that this is a special case of two concepts:

1. It is a single-perceptron and MLP (neural networks) are many of these combined (with slight modification).

2. The loss function derived is the binary cross-entropy between the output probability mass function $(p, 1-p)$ and the "one-hot" encoded label pmf $\tilde{y}$.

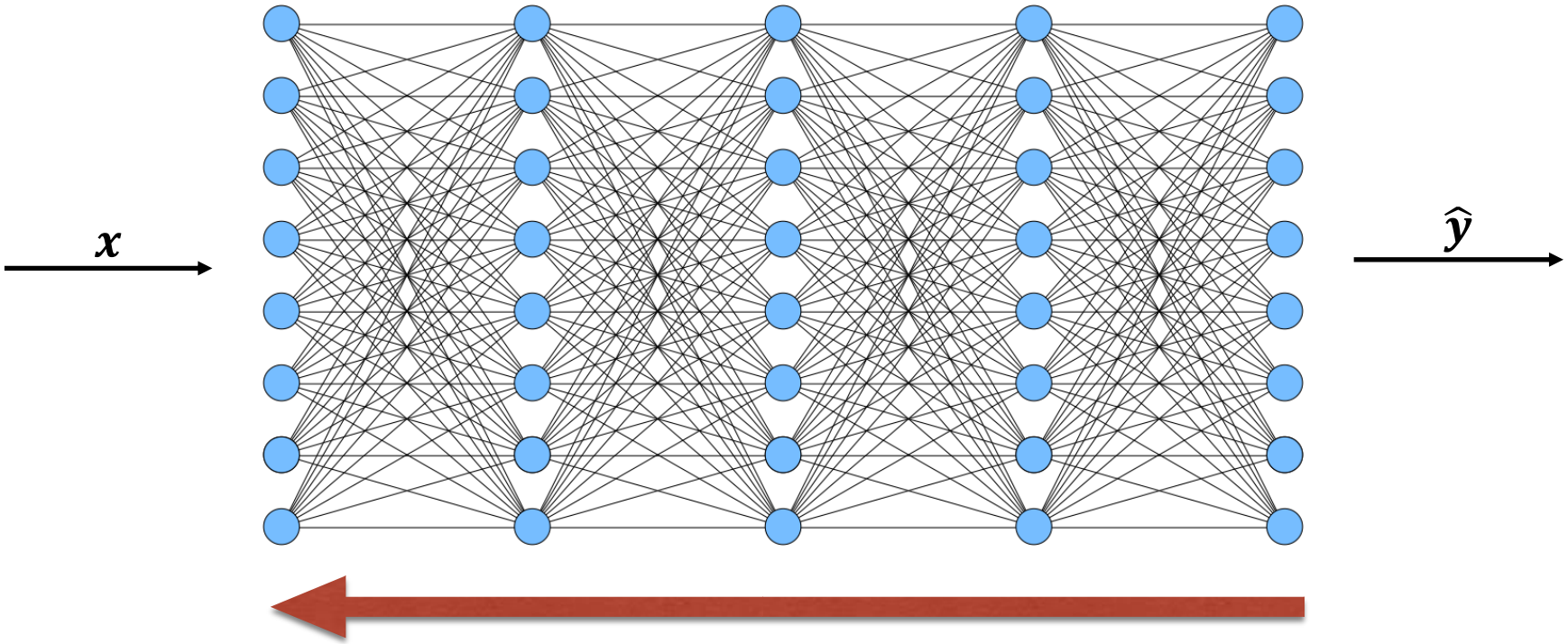
MULTILAYER PERCEPTRON NETWORKS (MLPS)

Forward propagation (inference and training)

$$\boldsymbol{a}^{(l)} = \underline{h}\left(\boldsymbol{W}_l \boldsymbol{a}^{l-1} + \boldsymbol{b}_l\right)$$

$$\Theta = \{\boldsymbol{W}_l, \boldsymbol{b}_l\}_{l=1}^{L}$$

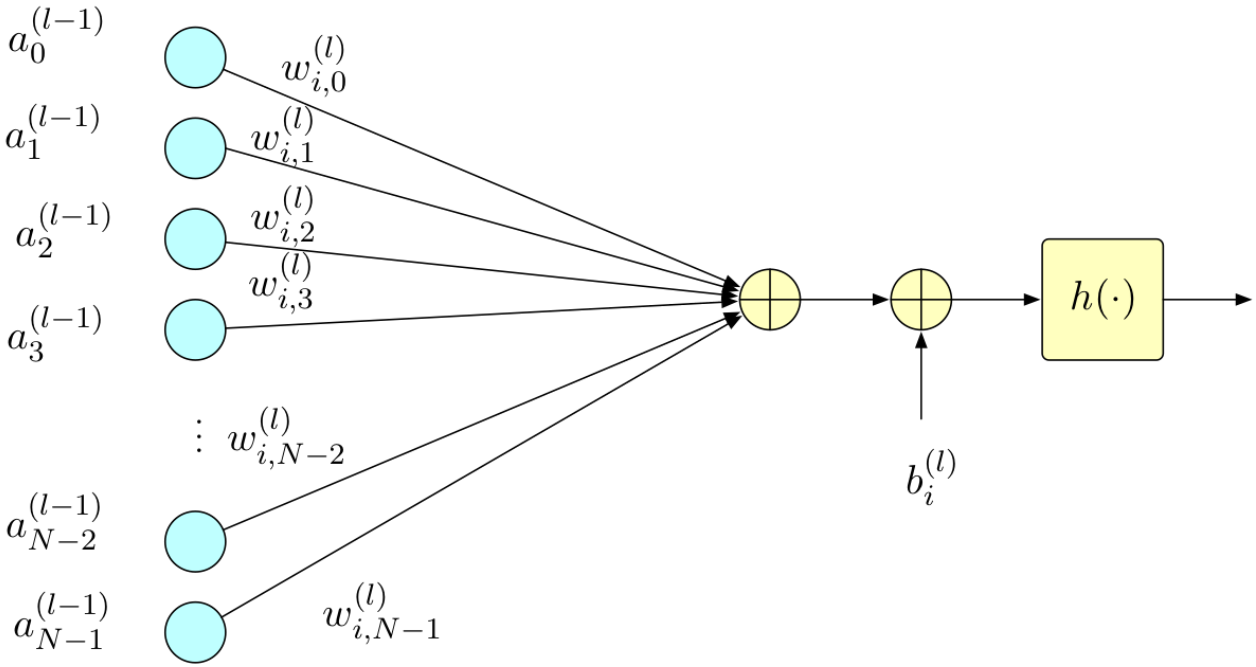
(trainable parameters)



Backward propagation (training)

Learn the trainable parameters using SGD and the chain-rule

MLP FORWARD PROPAGATION DETAILS



$$a_i^{(l)} = h\left(\left[\mathbf{w}_i^{(l)}\right]^{\mathrm{t}} \mathbf{a}^{(l-1)} + b_i^{(l)}\right)$$

processing at the i^{th} neuron (node) at layer l

look familiar?

BACK-PROPAGATION

MLP BACKPROP IDEA

do SGD on all trainable parameters:

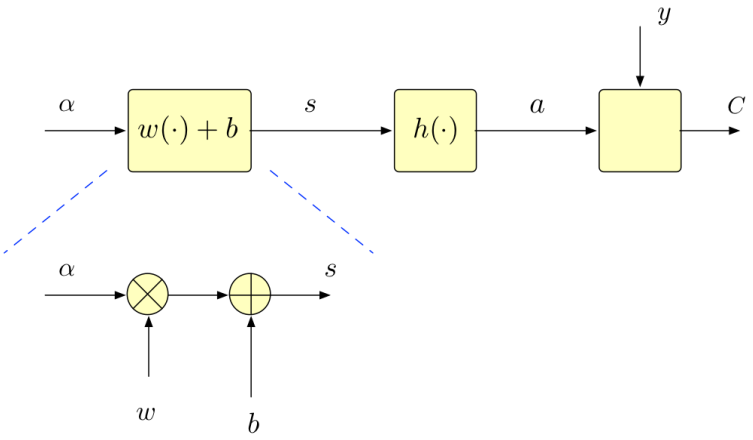
weight updates: $$w_{i,j}^{(l)} = w_{i,j}^{(l)} + \eta \frac{\partial C}{\partial w_{i,j}^{(l)}}$$

bias updates: $$b_i^{(l)} = b_i^{(l)} + \eta \frac{\partial C}{\partial b_i^{(l)}}$$

all layers, all indices: $$\forall l \in \{1,2,\dots,L\}; \forall i,j$$

Backprop is an algorithm for computing these, starting at the neural network output and propagating backward to the input layer using the **chain rule**

MLP BACKPROP: SCALAR EXAMPLE



$$\frac{\partial C}{\partial s} = \frac{\partial C}{\partial a}\frac{\partial a}{\partial s}$$

$$= \frac{\partial C}{\partial a}\frac{\partial h(s)}{\partial s}$$

$$= \dot{C}(a)\dot{h}(s)$$

$$\frac{\partial C}{\partial w} = \dot{C}(a)\dot{h}(s)\alpha$$

$$\frac{\partial C}{\partial b} = \dot{C}(a)\dot{h}(s)$$

example:

$$C(a) = \frac{1}{2}(y-a)^2$$

$$\dot{C}(a) = \frac{\partial C}{\partial a} = -(y-a)$$

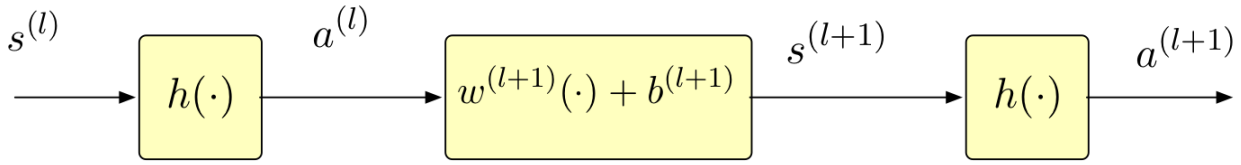
$$h(s) = \sigma(s) = \frac{1}{1+e^{-s}}$$

$$\dot{h}(s) = \frac{\partial h(s)}{\partial s} = \sigma(s)(1-\sigma(s))$$

$$w \leftarrow w + \eta a(1-a)(y-a)\alpha$$

$$b \leftarrow b + \eta a(1-a)(y-a)$$

MLP BACKPROP: SCALAR EXAMPLE



$$\delta^{(l)} \leftarrow \delta^{(l+1)}$$

$$\delta^{(l)} = \frac{\partial C}{\partial s^{(l)}} = \frac{\partial C}{\partial s^{(l+1)}} \frac{\partial s^{(l+1)}}{\partial s^{(l)}}$$

$$= \delta^{(l+1)} \frac{\partial s^{(l+1)}}{\partial a^{(l)}} \frac{\partial a^{(l)}}{\partial s^{(l)}}$$

$$= \delta^{(l+1)} w^{(l+1)} \dot{h}\left(s^{(l)}\right)$$

$$= \dot{h}\left(s^{(l)}\right) w^{(l+1)} \delta^{(l+1)}$$

$$= \dot{a}^{(l)} w^{(l+1)} \delta^{(l+1)}$$

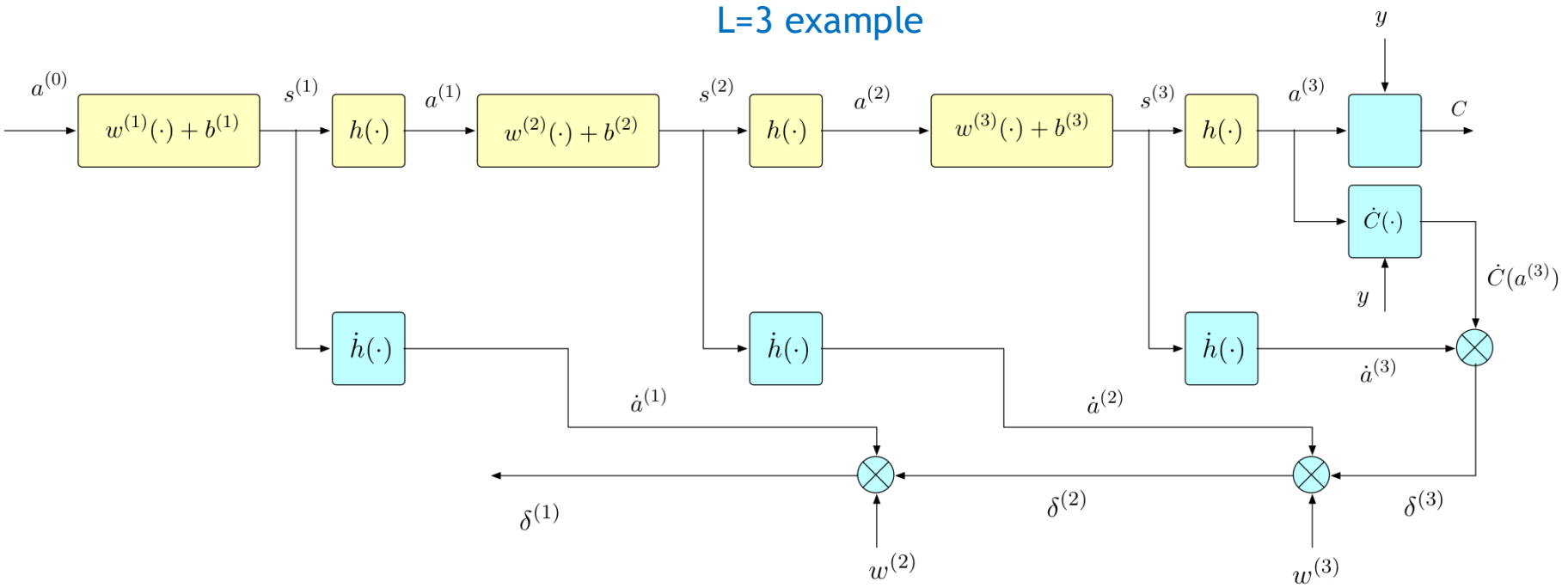
Shorthand:

$$\dot{a}^{(l)} \triangleq \dot{h}\left(s^{(l)}\right)$$

$$w^{(l)} \leftarrow w^{(l)} - \eta \delta^{(l)} a^{(l-1)}$$

$$b^{(l)} \leftarrow b^{(l)} - \eta \delta^{(l)}$$

MLP BACKPROP: SCALAR EXAMPLE



Note: update weights and biases only **after all** the deltas are computed

$$w^{(l)} \leftarrow w^{(l)} - \eta \delta^{(l)} a^{(l-1)}$$

$$b^{(l)} \leftarrow b^{(l)} - \eta \delta^{(l)}$$

$$\delta^{(l)} = \dot{a}^{(l)} w^{(l+1)} \delta^{(l+1)}$$

MLP BACKPROP: SUMMARY

$$\mathbf{a}_l = \text{act}(\mathbf{W}_l \mathbf{a}_{l-1} + \mathbf{b}_l) \qquad \text{(activations)}$$

$$\dot{\mathbf{a}}_l = \dot{\text{act}}(\mathbf{W}_l \mathbf{a}_{l-1} + \mathbf{b}_l) \qquad \text{(derivative activations)}$$

$$\boldsymbol{\delta}_L = \dot{\mathbf{a}}_L \odot \dot{\text{cost}}(\mathbf{y}, \mathbf{a}_L) \qquad \text{(delta initialization)}$$

$$\boldsymbol{\delta}_l = \dot{\mathbf{a}}_l \odot \left[\mathbf{W}_{l+1}^T \boldsymbol{\delta}_{l+1}\right] \qquad \text{(delta recursion)}$$

$$\mathbf{W}_l \leftarrow \mathbf{W}_l - \eta \boldsymbol{\delta}_l \mathbf{a}_{l-1}^T \qquad \text{(weight SGD update)}$$

$$\mathbf{b}_l \leftarrow \mathbf{b}_l - \eta \boldsymbol{\delta}_l \qquad \text{(bias SGD update)}$$

specialized to vectorized output activation and additive cost

USING BATCHES

For each data sample:

$$(\mathbf{x}[n], \mathbf{y}[n])$$

Do FF and BP to compute activations, a-dots, and deltas

$$\mathbf{a}_l[n] = \text{act}(\mathbf{W}_l \mathbf{a}_{l-1}[n] + \mathbf{b}_l)$$

$$\dot{\mathbf{a}}_l[n] = \dot{\text{act}}(\mathbf{W}_l \mathbf{a}_{l-1}[n] + \mathbf{b}_l)$$

$$\boldsymbol{\delta}_L[n] = \dot{\mathbf{a}}_L[n] \odot \dot{\text{cost}}(\mathbf{y}[\mathbf{n}], \mathbf{a}_L[n])$$

$$\boldsymbol{\delta}_l[n] = \dot{\mathbf{a}}_l[n] \odot \left[\mathbf{W}_{l+1}^T \boldsymbol{\delta}_{l+1}[n]\right]$$

Do one SGD update after finishing the batch

$$\mathbf{W}_l \leftarrow \mathbf{W}_l - \eta \frac{1}{B} \sum_{n=1}^{B} \boldsymbol{\delta}_l[n] \mathbf{a}_{l-1}^T[n]$$

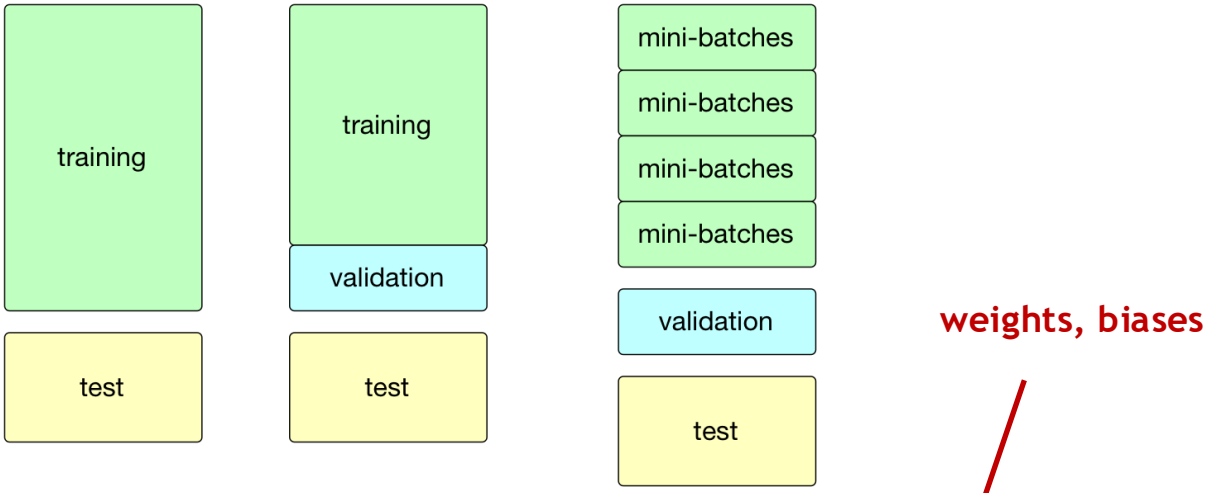
$$\mathbf{b}_l \leftarrow \mathbf{b}_l - \eta \frac{1}{B} \sum_{n=1}^{B} \boldsymbol{\delta}_l[n]$$

USC Viterbi
School of Engineering

DEALING WITH DATA

TRAIN VS. TEST

TRAINING (+ VAL)/TEST SPLIT



weights, biases

(training-)training: use this for SGD learning — trainable parameters

(training-)validation: use this for SGD learning — hyper-parameters

test: only use this when you are done to verify

learning rate, batch size, etc.

TRAINING (+ VAL)/TEST SPLIT

Example

100,000 total (x_n, y_n) (shuffle it all once)

70,000 train
15,000 validate
15,000 test

Suppose: batch size = 70:

1000 mini-batches in the training data (1000 iterations per epoch)

1000 gradient updates in an epoch, each averaged over 70 samples

these are typically processed serially: batch 1, batch 2, etc.

gradient updates are serial

(can change with many parallel compute nodes)

run inference (forward only) on validation data after each epoch

monitor learning curve, iteration hyper-hyper-parameter search…

when done, run on test

EVALUATION METRICS

- Accuracy

$$= \frac{TP + TN}{TP + TN + FP + FN}$$

- Sensitivity / Recall

$$= \frac{TP}{TP + FN}$$

- Specificity

$$= \frac{TN}{TN + FP}$$

- Precision

$$= \frac{TP}{TP + FP}$$

- F1-Score

$$= 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

- J-Score

$$= Sensitivity + Specificity - 1$$

- False Positive Rate

$$= 1 - Specificity$$

- Area under the ROC curve (AUC)

$$= \frac{S_p - n_p(n_n + 1)/2}{n_p\, n_n}$$

DESIGNING DATASETS

PRINCIPLES FOR DESIGNING DATASETS

"Neural Networks are Lazy"

is ***the*** principle that should guide your dataset design

You want to maximize the ***coverage*** in your dataset

e.g., cats with non-green backgrounds were not covered in previous example

- Include maximum diversity in your dataset
- Think lazy like a neural network and design your dataset for maximum coverage
- Include difficult and extremely difficult examples in your dataset (even if you have to create them!)
  - Giving tough examples will not make your trained network worse at the easy cases!
- You can never have too much (valid) data

PRINCIPLES FOR DESIGNING DATASETS

Collection	find data online, crowdsource, scrape online resources
Labeling	labor intensive task (ways around this: synthetic data, use ML to label, $$$)
Coverage	make your neural network work (not be lazy) by giving examples of every scenario you expect it to work in
Contamination	accept that there will be mislabeled data in huge datasets due to human error or ambiguities
Cleaning	correct contamination effects, remove misleading or ambiguous examples. Automate this.
Augmentation	use synthetic means to produce better coverage and more difficult examples from your baseline data. In some cases, you may create synthetic data to augment your data

PRINCIPLES FOR DESIGNING DATASETS

In practice, designing your dataset is the most important aspect in developing a deep-learning solution

90% of effort is spent on dataset design and maintenance (my experience)

this is not apparent from reading papers and books because most materials focus on using publicly available datasets that serve as test benchmarks

in our class, we are crowd-sourcing project datasets to try to illustrate the practical issues, but still not at a practical scale

TYPICAL FLOW FOR DEEP LEARNING DEVELOPMENT

General Good Practice

Get a good **baseline:**

solve the problem without a neural network first if possible

use a published baseline network as a baseline if you know the problem requires deep learning

keep your training simple to start

overfit a subset of data to make sure all is working before going big

Develop a **Full Dev Pipeline (scripts):**

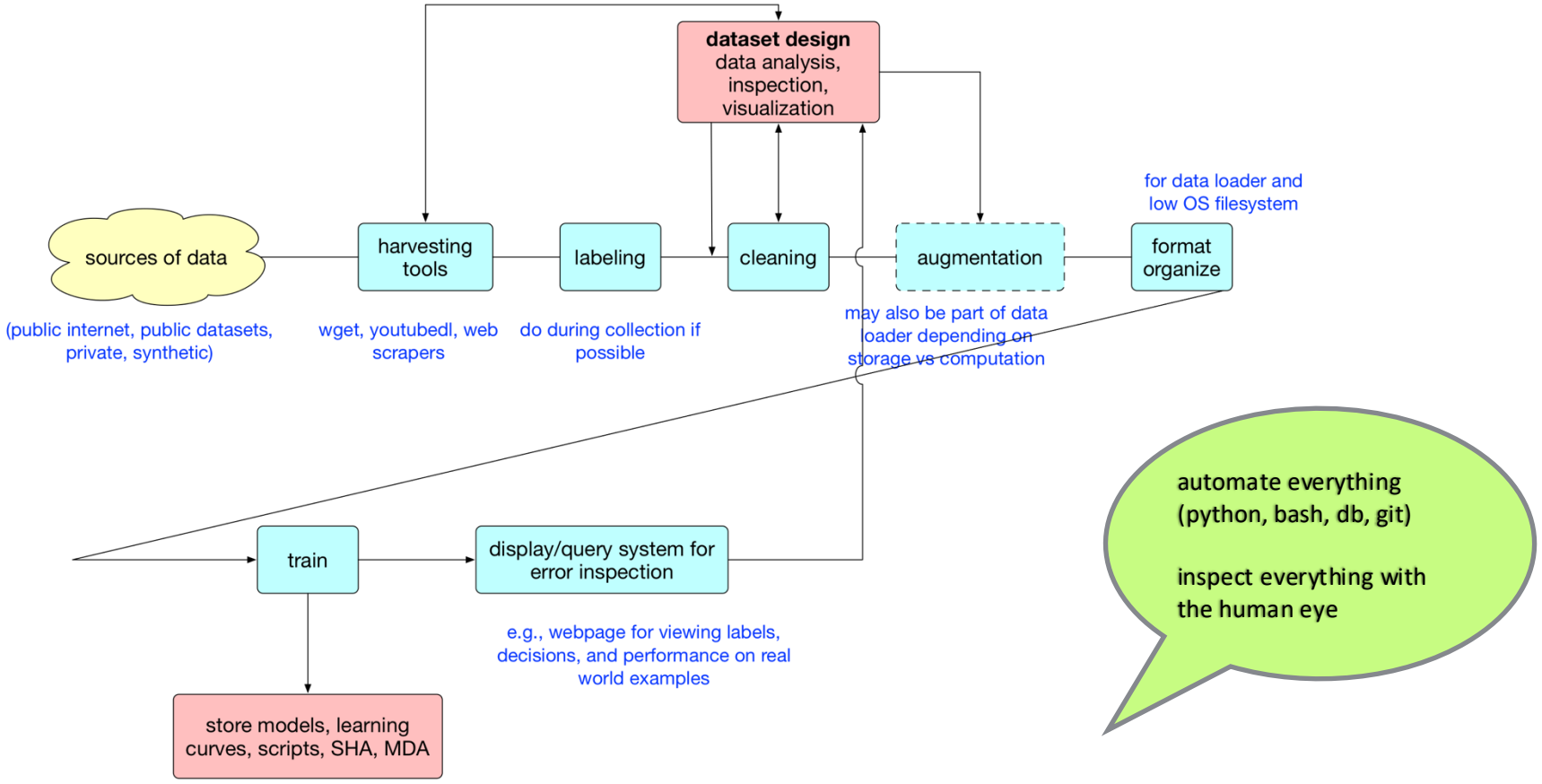
viewing/interpreting datasets and examples

cultivating/updating your dataset

training with version control and auto-documentation

testing and visualization of the result on real-world data

TYPICAL FLOW FOR DEEP LEARNING DEVELOPMENT



popular blog by Telsa Sr. Director of AI hits many of these same points

COMMON DATA NORMALIZATION METHODS

feature-wise standardization

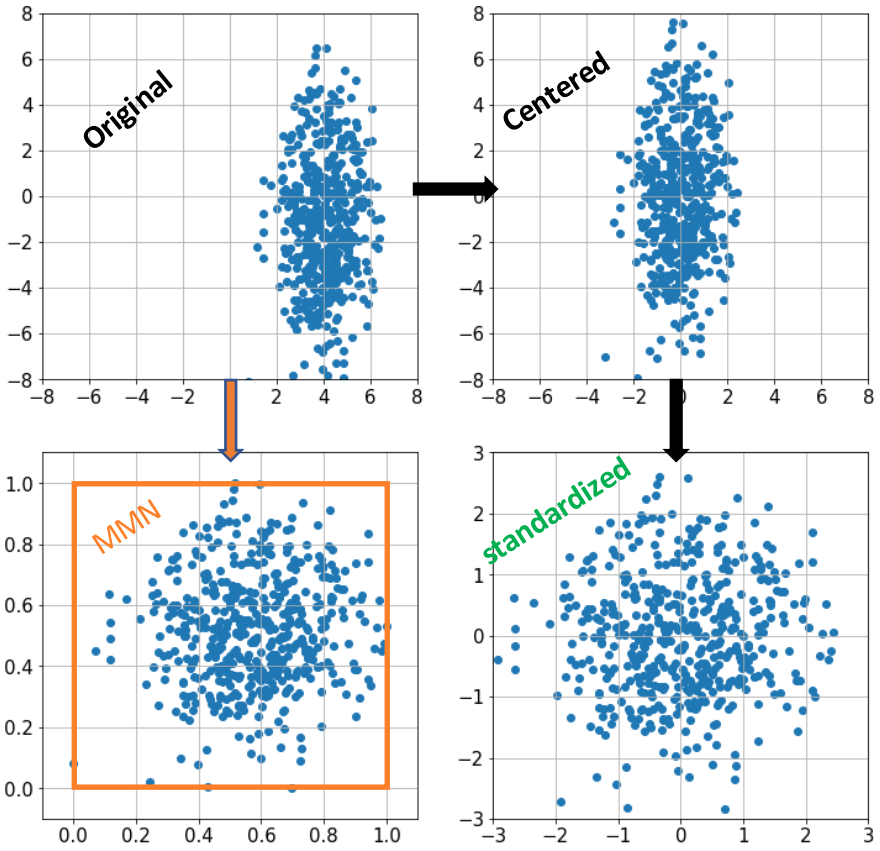
$$v_n[i] = \frac{x_n[i] - \hat{m}_x[i]}{\hat{\sigma}_x[i]}$$

scale each dimension of feature vector to mean 0, variance 1

feature-wise "minmax" normalization

$$v_n[i] = \frac{x_n[i] - \min_n x_n[i]}{\max_n x_n[i] - \min_n x_n[i]}$$

scale each dimension of feature vector to range [0,1]



AUGMENTATION

increase the diversity and/or difficulty of your training data through pre-processing

Examples (images):

rotate

flip

reflect

"cut-out"
(remove patches)

translate

add noise

camera
modeling

blur (change resolution)

Will see in CNNs that there are some nice built in image augmentation tools in python

Examples (audio):

add noise

add reverb

filter/equalize

resample
(change sample rates)

mic/speaker modeling

PRE-PROCESSING (PCA, LDA, NORMALIZATION)

Use statistics collected from **Training Data** only

apply same transformation to training, val, test data

Note that many of these techniques can be viewed as a fixed linear layer at the start of the network that is not trained as part of BP

alternative is to have a “bottleneck" layer for dimensionality reduction (*i.e.*, learn ~ LDA during BP)

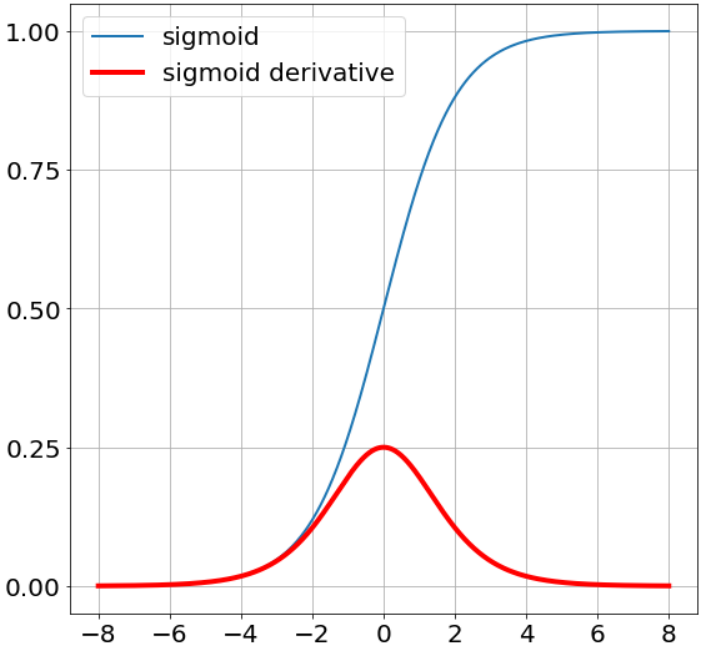
batch-normalization similarly is a method of learning normalization

VANISHING GRADIENT

VANISHING GRADIENT PROBLEM SQUASHING ACTIVATIONS

$$\sigma'(x) = \sigma(x)\,(1 - \sigma(x))$$

the maximum value of $\dot{\sigma}(\cdot)$ is 0.25…



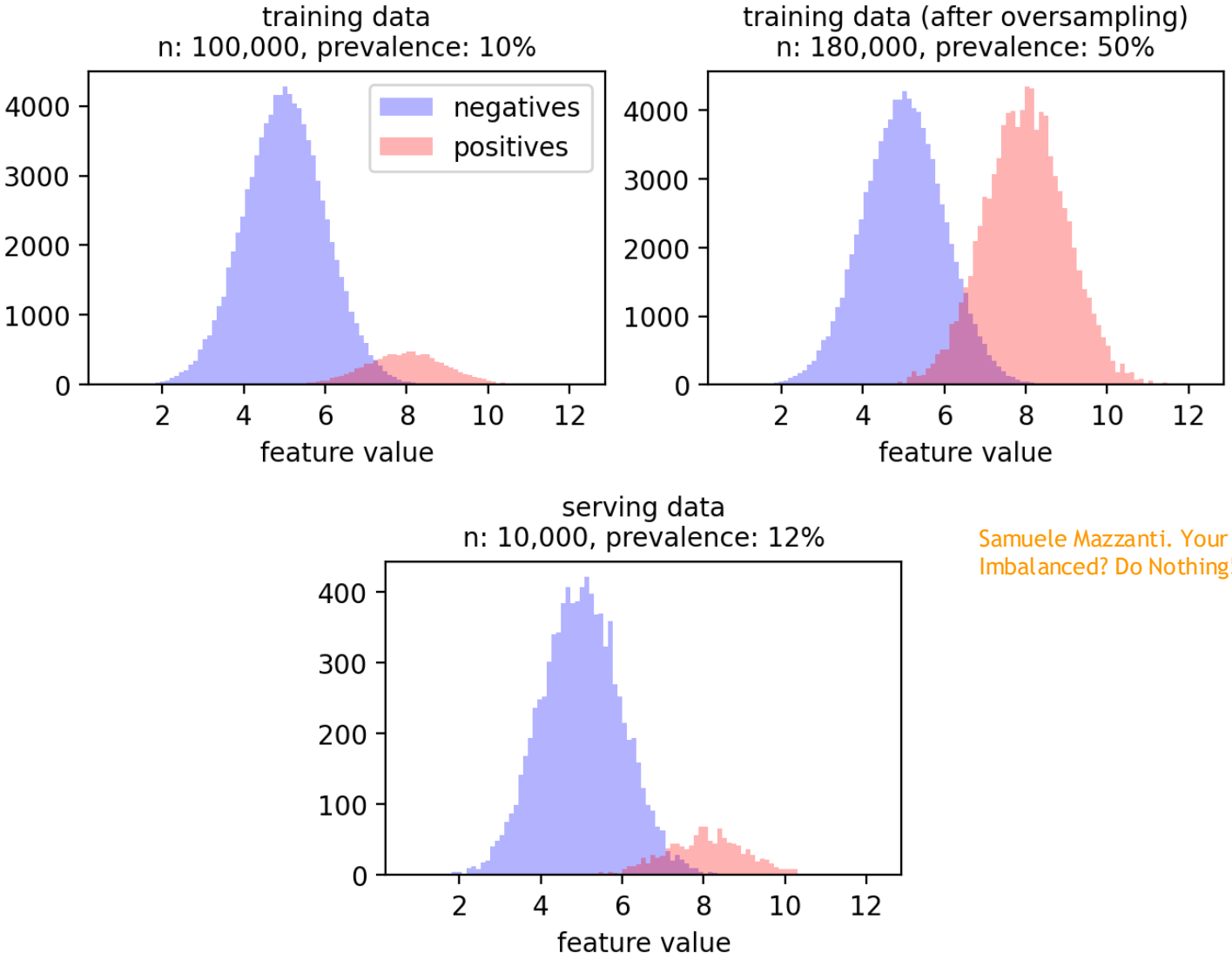
$$\boldsymbol{\delta}_1 = \left(\dot{\sigma}(\mathbf{s}_1) \odot \left[\mathbf{W}_2^{\mathrm{t}}\boldsymbol{\delta}_2\right]\right)\left(\dot{\sigma}(\mathbf{s}_2) \odot \left[\mathbf{W}_3^{\mathrm{t}}\boldsymbol{\delta}_3\right]\right)\left(\dot{\sigma}(\mathbf{s}_3) \odot \left[\mathbf{W}_4^{\mathrm{t}}\boldsymbol{\delta}_4\right]\right)\left(\dot{\sigma}(\mathbf{s}_4) \odot \left[\mathbf{W}_5^{\mathrm{t}}\boldsymbol{\delta}_5\right]\right)\left(\dot{C}(\mathbf{y}, \mathbf{a}_5) \odot \dot{\sigma}(\mathbf{s}_5)\right)$$

… AND EXPLODING GRADIENT PROBLEM

$$w = w + \eta \frac{\partial C}{\partial w} \qquad \left|\frac{\partial C}{\partial w}\right| \gg 1$$

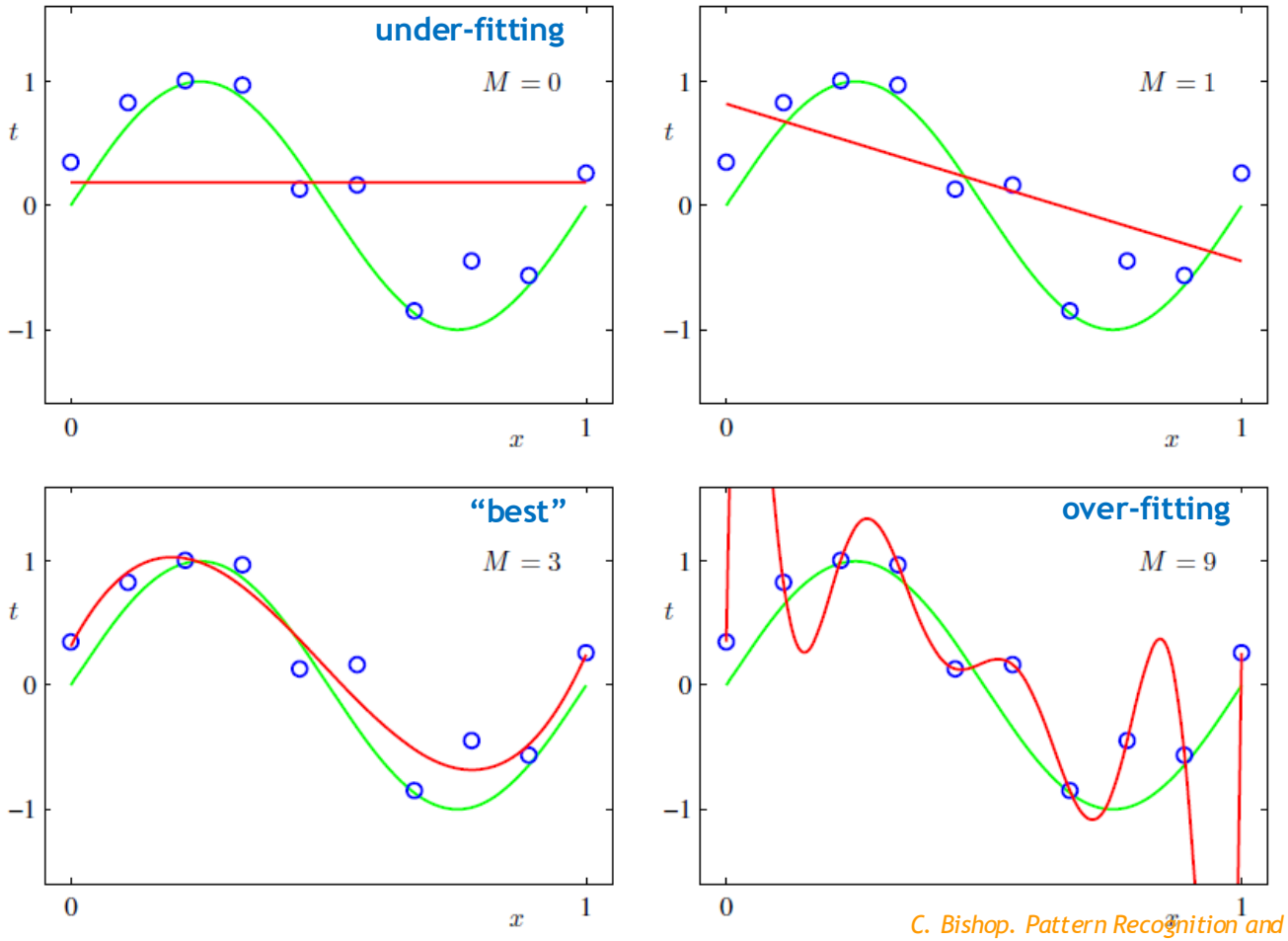
Behavior as if learning-rate too large, sensitive to high frequency noise

IMBALANCED DATA PROBLEM



Samuele Mazzanti. Your Dataset Is Imbalanced? Do Nothing!

COMPLEX HYPOTHESIS CLASS LEADS TO OVER-FITTING



Bias-Variance trade-off (EE 503)

$\text{MSE} = \text{V} + \text{Bias}^2$

C. Bishop. Pattern Recognition and Machine Learning. 2006.

Choosing the right model (hypothesis) is challenging

PYTORCH

INTRODUCTION TO PYTORCH

Example: use the "Functional API" to define the model

```python
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms
import torch.nn.functional as F
import matplotlib.pyplot as plt
import numpy as np

train_set = torchvision.datasets.FashionMNIST(root = "./data", train = True, download = True,
                                              transform = transforms.ToTensor())
test_set = torchvision.datasets.FashionMNIST(root = "./data", train = False, download = True,
                                             transform = transforms.ToTensor())

train_loader = torch.utils.data.DataLoader(train_set, batch_size=100, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_set, batch_size=100, shuffle=False)

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.hidden = nn.Linear(num_pixels, 128)
        self.output = nn.Linear(128, 10)

    def forward(self, x):
        x = F.relu(self.hidden(x))
        x = self.output(x)
        return x

model = Net()

loss_func = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.0001)

num_epochs =
for epoch in range(num_epochs):
    [...]

```

define a torch.nn network (modular)

PYTORCH — TRAINING LOOP (NO VALIDATION)

```
num_epochs = 4
count = 0
for epoch in range(num_epochs):
    correct = 0
    for images, labels in train_loader:
        count += 1
        input = images.view(-1, 28*28)

        # forward pass
        outputs = model(input)
        loss = loss_func(outputs)

        # backprop
        optimizer.zero_grad()
        loss.backward()

        # optimize
        optimizer.step()

        # not normally in training
        predictions = torch.max(outputs, 1)[1]
        correct += (predictions == labels).sum().numpy()

    print(f'Epoch: {epoch+1:02d}, Iteration: {count:5d}, Loss: {loss.data:.4f}, ' +
          f'Accuracy: {100 * correct/len(train_loader.dataset):2.3f}%')

print('Finished Training')
```

```
Epoch: 01, Iteration:   600, Loss: 0.7919, Accuracy: 69.465%
Epoch: 02, Iteration:  1200, Loss: 0.5634, Accuracy: 80.110%
Epoch: 03, Iteration:  1800, Loss: 0.2925, Accuracy: 82.467%
Epoch: 04, Iteration:  2400, Loss: 0.4984, Accuracy: 83.615%
Finished Training
```

ACTIVATIONS IN PYTORCH

$$\mathbf{h}(\mathbf{s}) = \frac{1}{\sum_{m=0}^{M-1} e^{s_m}} \begin{bmatrix} e^{s_0} \\ e^{s_1} \\ \vdots \\ e^{s_{M-1}} \end{bmatrix}$$

soft-max:
produces $M \times 1$ probability mass function use for M-ary classification between mutual exclusive classes

$$\sigma(s) = \frac{1}{1 + e^{-s}}$$

sigmoid:
produces probability of "class 1" for a binary classification test

binary classification:

1 output neuron with sigmoid and BCE

vs.

2 output neurons with softmax and MCE

WEIGHT (AND BIAS) INITIALIZATION

$$\theta \leftarrow \theta - \eta \frac{\partial C}{\partial \theta}$$

what do we initialize parameter theta with?

empirical observation: some initializations are better than others

zero initialization?

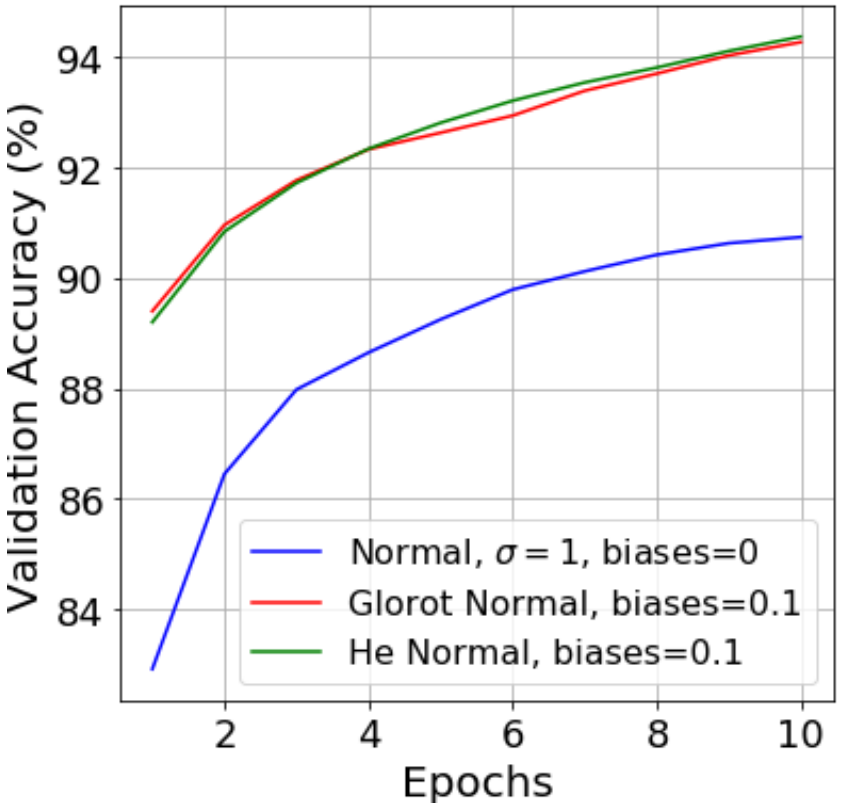
all linear activations are 0…

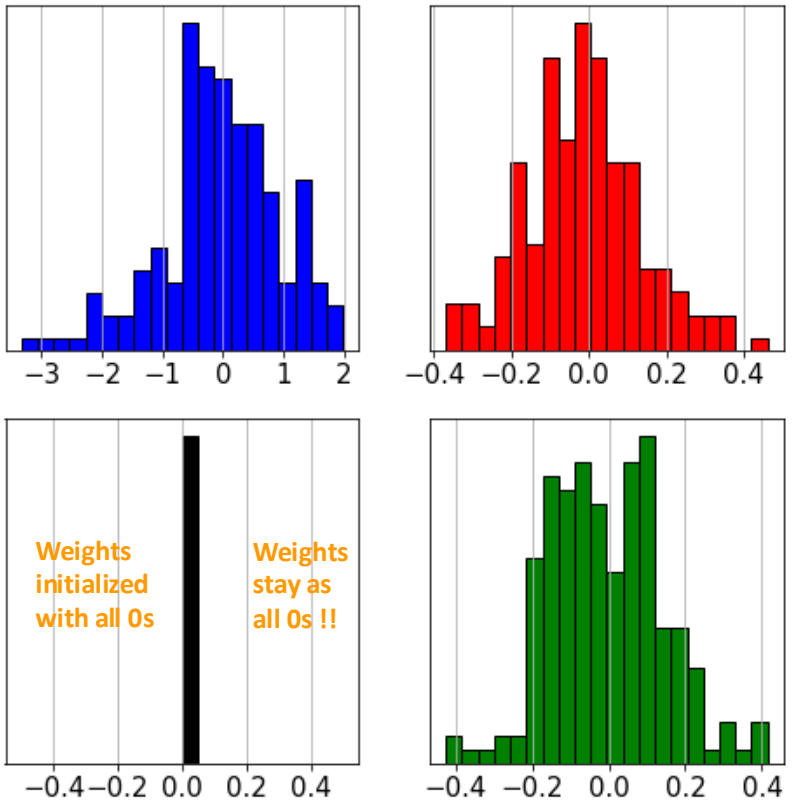
the deltas will be 0 too…

$$\boldsymbol{\delta}_l = \dot{\mathbf{a}}_l \odot \left[\mathbf{W}_{l+1}^{\mathrm{t}} \boldsymbol{\delta}_{l+1} \right]$$

use random initialization…

COMPARISON OF INITIALIZERS





MNIST [784,200,10]
Regularization: None

Histograms of a few weights in 2nd junction after training for 10 epochs

COST (LOSS) FUNCTIONS

LOSS FUNCTIONS — L2 FOR REGRESSION

$$C = \|\mathbf{y} - \mathbf{a}\|_2^2 = \sum_{i=1}^{M} (y_i - a_i)^2$$

(squared) L2 norm of error
or sum of squared error

$$C = \frac{1}{M}\|\mathbf{y} - \mathbf{a}\|_2^2 = \frac{1}{M}\sum_{i=1}^{M} (y_i - a_i)^2$$

average squared error

these are equivalent

PyTorch implements the mean by default, see options
(good since it is normalized for number of classes)

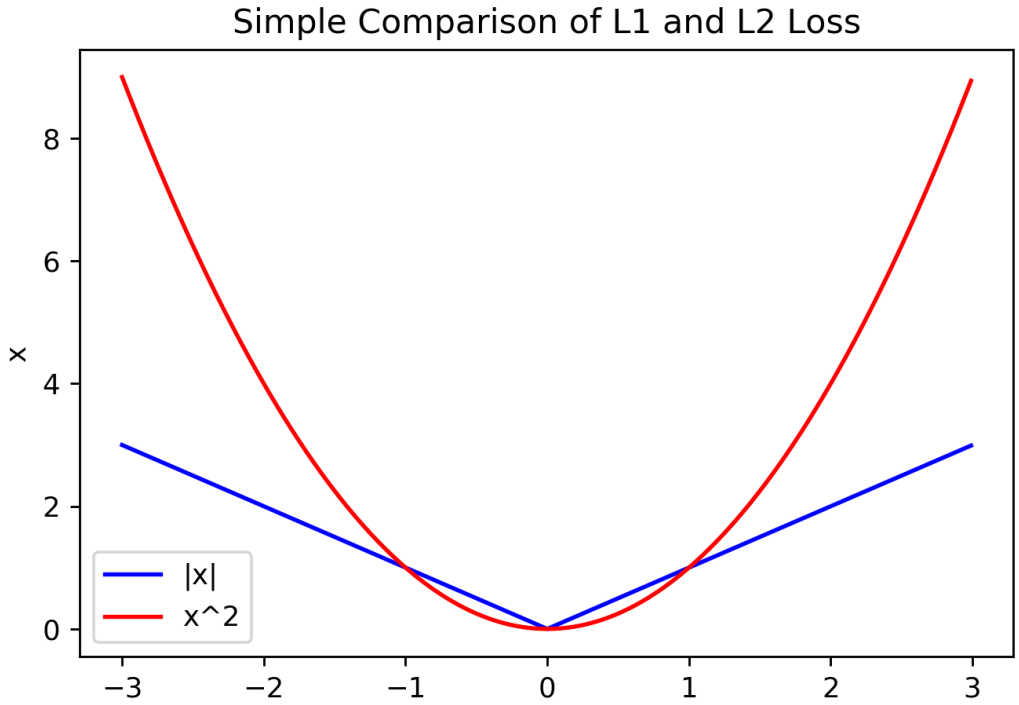
```
ms = nn.MSELoss()
output = ms(
    torch.FloatTensor([[1, 1, 1], [2, 2, 2]]),
    torch.FloatTensor([[0, 0, 0], [3, 3, 3]]))
```

```
tensor(1.)
```

for BP Initialization

$$\frac{d}{da}(y - a)^2 = 2(y - a)$$

LOSS FUNCTIONS — L1 VS L2



L2 penalizes large error more than L1

L2 corresponds to power/energy for ECE

L1 will typically induce sparsity in your weights - allows a few large weights and many other weights are near 0

LOSS FUNCTIONS — MULTICATEGORY CROSS ENTROPY

Recall with one-hot (hard labels)

$$C = -\sum_{i=1}^{M} y_i \ln a_i \longrightarrow C = -\ln a_m$$

Class m is true

```
loss_func = nn.CrossEntropyLoss()
loss_func(
    torch.tensor([
        [np.log(0.9),  np.log(0.05), np.log(0.05)],
        [np.log(0.05), np.log(0.89), np.log(0.06)],
        [np.log(0.05), np.log(0.01), np.log(0.94)]]
    ),
    torch.tensor([0,1,2])
).numpy()
```

```
array(0.09458991)
```

```
(np.log(0.9) + np.log(0.89) + np.log(0.94)) / 3
= -0.094590
```

(averaged over batch size)

recall: MCE is the negative log-likelihood (NLL) with regression error model:

$$P(class = i) = a_i$$

COST (LOSS) FUNCTIONS — BINARY CROSS ENTROPY

for $M = 2$ outputs — binary classification

$$C = -y\ln(a) - (1-y)\ln(1-a) = y\ln\left(\frac{1}{a}\right) + (1-y)\ln\left(\frac{1}{1-a}\right)$$

Same as MCE with $a_0 = a,\ a_1 = 1 - a$

PyTorch uses this

```
nn.BCELoss()(
    torch.tensor([[.6, .8, .1]]),
    torch.tensor([[0., 1., 0.]])
)
```

```
tensor(0.4149)
```

```
def bce(y,a):
    return -1*y*np.log(a+1e-10) -(1-y)*np.log(1-a+1e-10)

np.mean(bce(np.array([0,1,0]), np.array([0.6, 0.8, 0.1])))
0.414932
```

Compare with `nn.BCEWithLogitsLoss()`

WEIGHT REGULARIZATION

REGULARIZATION

What is regularization and why do it?

We have seen an example: enforce penalty on weights to bias toward a prior distribution.

effect is to reduce over-fitting (get smaller weights)

Not all regularization methods can be viewed this way

in some cases, intuitive, empirical penalty enforcing functions are used

What is a more general definition of regularization?

regularization is anything you do in training that is aimed at improving generalization over accuracy — *i.e.*, anything that does not optimize the cost on the training data

we will see very different versions of this — *e.g.*, drop-out

REGULARIZATION INTERPRETATION

$$\max_{\theta} p_{y(t)|\boldsymbol{x}(t),\Theta(t)}(y|\boldsymbol{x},\theta)p_{\Theta(t)|\boldsymbol{x}(t)}(\theta|\boldsymbol{x}) \quad \Leftrightarrow \quad \min_{\boldsymbol{w}} \|\boldsymbol{y}-\boldsymbol{X}\boldsymbol{w}\|^2 + \lambda\|\boldsymbol{w}\|^2$$

The *a-priori* Gaussian distribution on the weights leads to "L2 regularization"

penalizes large $\boldsymbol{w}$ — even if large $\boldsymbol{w}$ cause smaller squared error

this can be viewed a method to combat **over-fitting**

λ is called the regularization coefficient in this context

Larger $\lambda \rightarrow$ penalize larger weights more aggressively (at expense of SE)

REGULARIZERS

$$\lambda \approx \frac{\text{Importance of small weights}}{\text{Importance of minimizing training loss}}$$

$\lambda = 0$ → $w^* \sim \arg\min C_{\text{no-reg}}(\mathbf{w})$

could be over-fitting, depends on capacity of model, dataset properties, and inference problem

$\lambda = \infty$ → $w^* \sim 0$

under-fitting

Typically: $10^{-5} \lesssim \lambda \lesssim 10^{-3}$

REGULARIZERS IN PYTORCH

https://pytorch.org/docs/stable/optim.html

CLASS `torch.optim.SGD(params, lr=<required parameter>, momentum=0, dampening=0, weight_decay=0, nesterov=False)` [SOURCE]

Implements stochastic gradient descent (optionally with momentum).

Nesterov momentum is based on the formula from On the importance of initialization and momentum in deep learning.

Parameters

- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups
- **lr** (*float*) – learning rate
- **momentum** (*float, optional*) – momentum factor (default: 0)
- **weight_decay** (*float, optional*) – weight decay (L2 penalty) (default: 0)
- **dampening** (*float, optional*) – dampening for momentum (default: 0)
- **nesterov** (*bool, optional*) – enables Nesterov momentum (default: False)

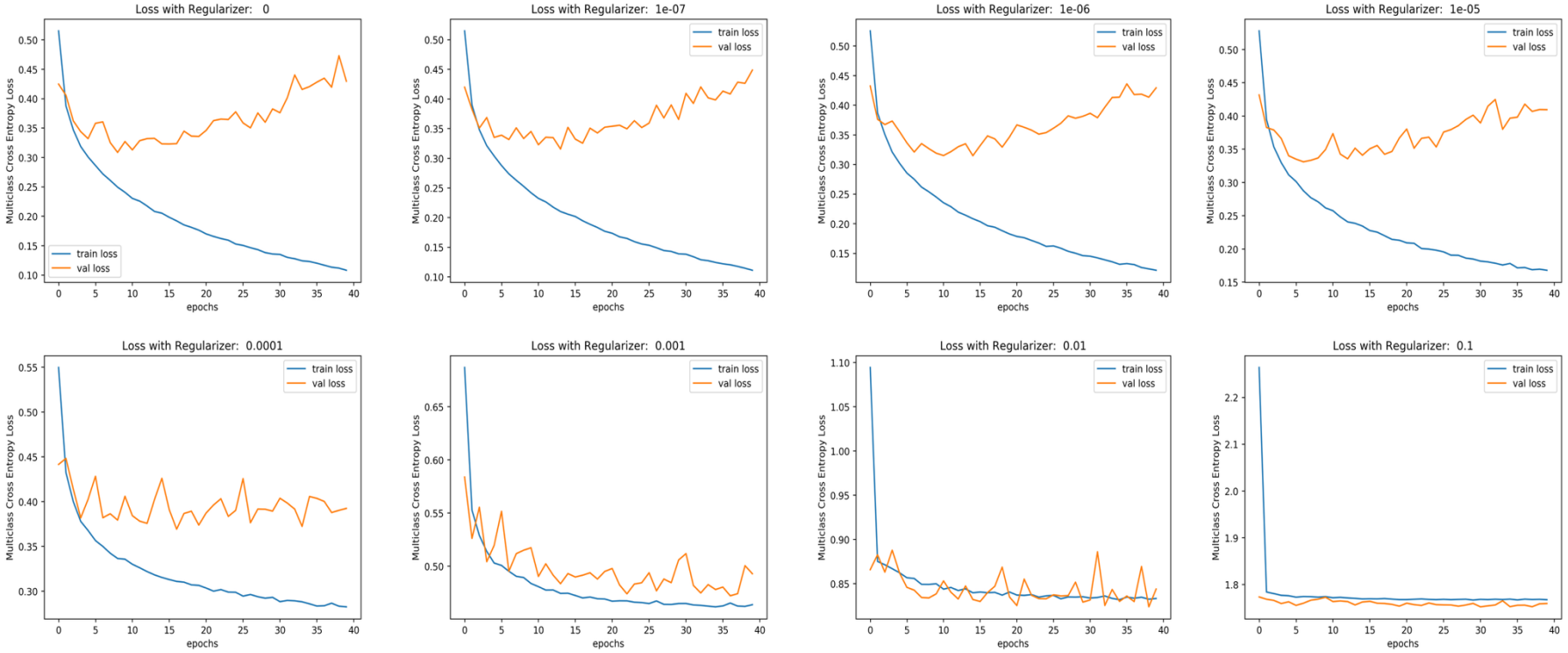
```
optim.SGD([
    {'params': model.base.parameters(), 'weight_decay': 0.},
    {'params': model.classifier.parameters(), 'weight_decay': 0.01}
], lr=1e-2)
```

Use per-parameter options for more control

Most optimizers include a **`weight_decay`** parameter
L^2 penalty, default = 0

works with autograd package

LET'S TRY L2 REGULARIZATION…



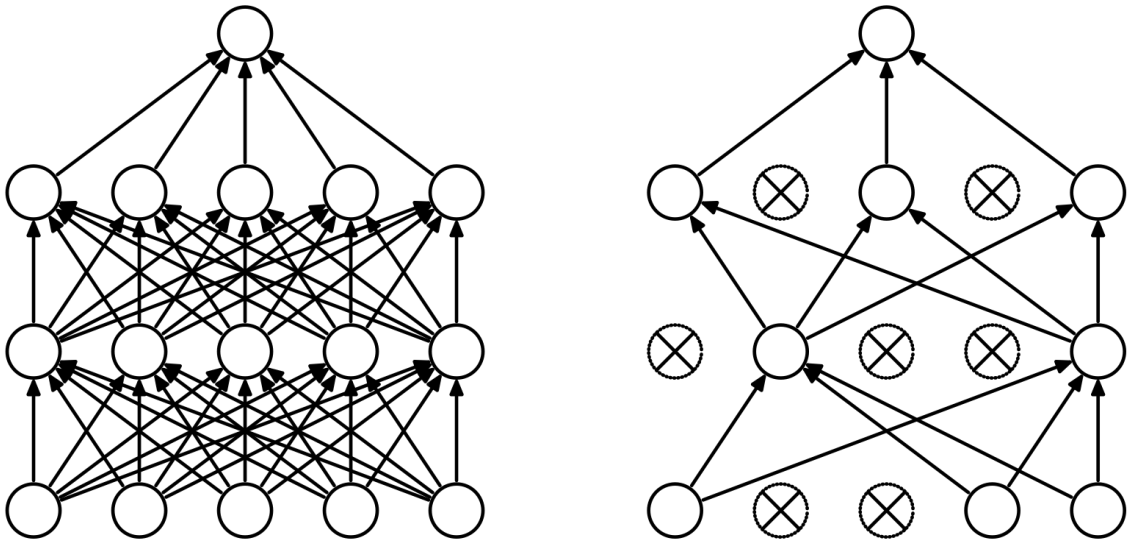
just using regularization, we need λ~1e-3 to prevent over-fitting, but the loss is much higher (~0.45 vs 0.1)

DROPOUT REGULARIZATION

DROPOUT — A DIFFERENT TYPE OF REGULARIZATION

remove nodes in a layer with some dropout probability/rate

the **random pattern** is generated at the start of each mini-batch and held fixed during that mini-batch



(a) Standard Neural Net

(b) After applying dropout.

N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," Journal of Machine Learning Research, vol. 15, pp. 1929–1958, 2014

DROPOUT

very effective at reducing over fitting and improving generalization

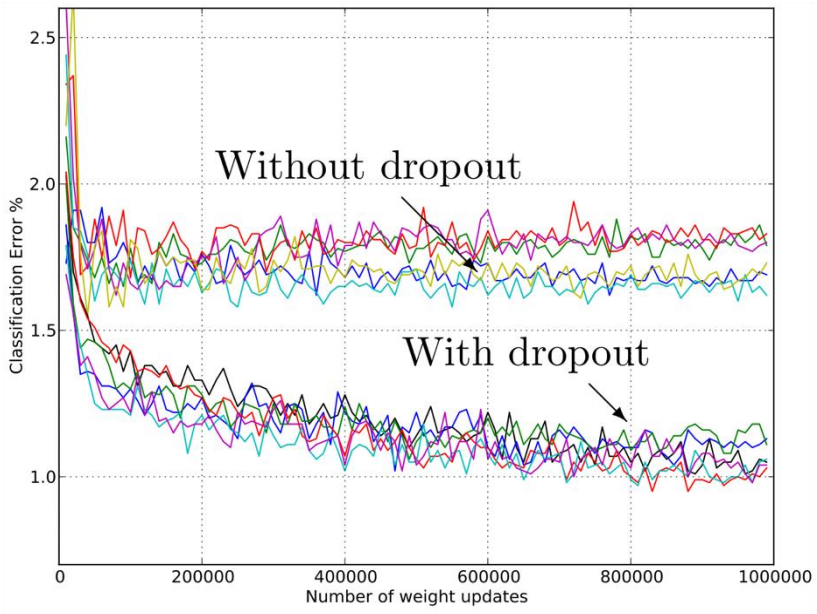
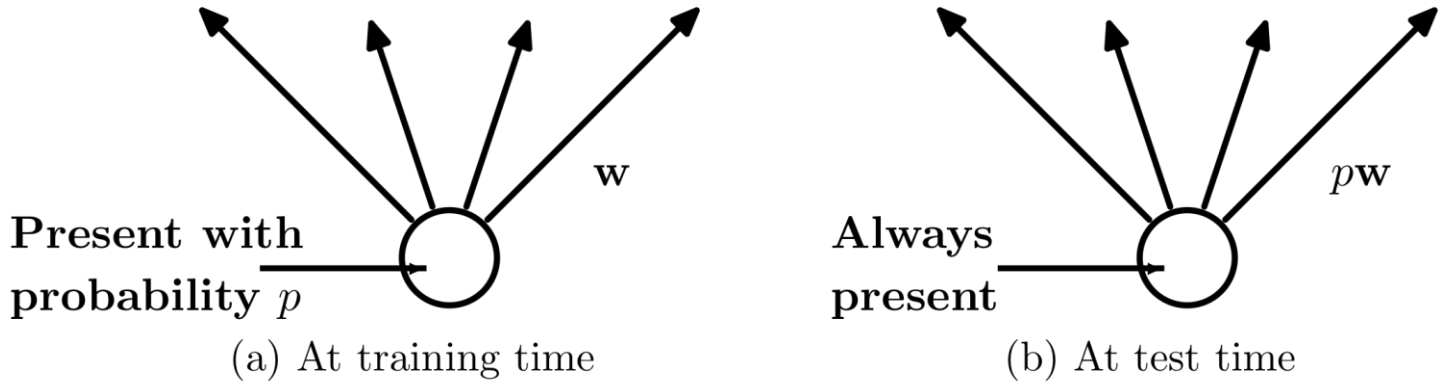


Figure 4: Test error for different architectures with and without dropout. The networks have 2 to 4 hidden layers each with 1024 to 2048 units.

N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," Journal of Machine Learning Research, vol. 15, pp. 1929-1958, 2014

DROPOUT — ONLY DURING TRAINING!

Dropout is **used during training**, but in inference mode, all nodes are present



(a) At training time (b) At test time

for inference, replace the trained weights with $p \cdot w$, where $(1 - p)$ is the dropout rate

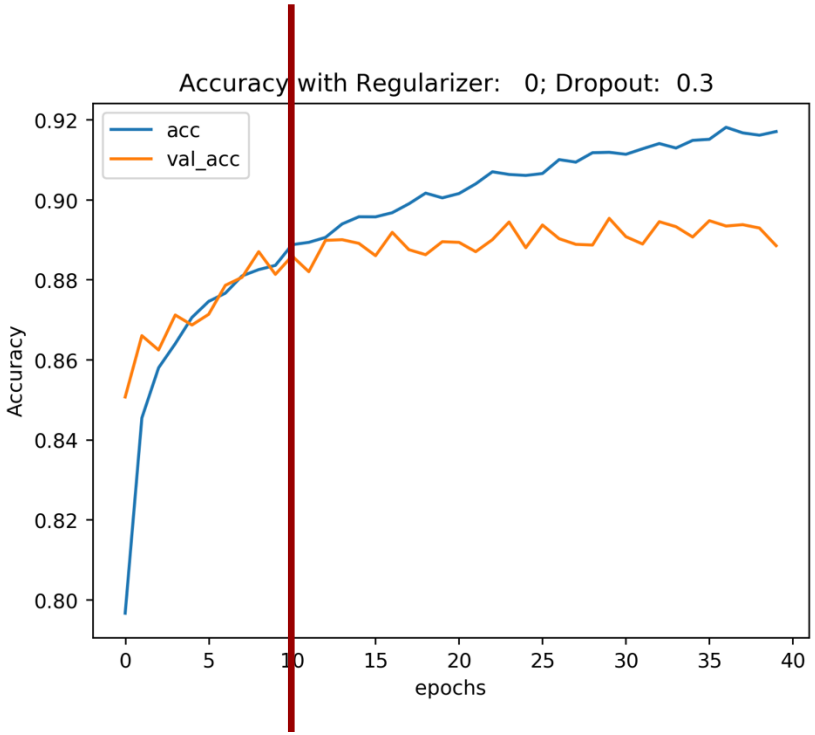
(ad hoc due to nonlinearities, but it works well enough)

N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," Journal of Machine Learning Research, vol. 15, pp. 1929–1958, 2014

ANOTHER REGULARIZATION METHOD

"early stopping"

stop training when val starts performs *consistently* better than train



stop at ~10 epochs

OPTIMIZERS

OPTIMIZERS

Optimizers are **modifications** to standard Stochastic Gradient Descent (SGD)

Three common modifications:

1. Gradient filtering
2. Gradient normalization
3. Learning rate schedule

- 1 and 2 usually associated with the "optimizer"
- learning rate schedule considered a separate parameter tuning task

SUMMARY OF OPTIMIZERS

	gradient filtering	gradient normalization	grad variance filter	learning rate schedule
SGD	none	none	n/a	separate
SGD w/ momentum	AR1, unit input gain	none	n/a	separate
SGD w/ Nesterov Momentum	ARMA1 (1 pole, 1 zero)	none	n/a	separate
Adagrad	none	yes	summer	separate, but gradient norm does alter
Adadelta	none	yes	AR1, unit DC gain	separate, but gradient norm does alter
RMSprop	none	yes	AR1, unit DC gain	separate, but gradient norm does alter
Adam	AR1, unit input gain, transient compensation	yes	AR1, unit input gain, transient compensation	separate, but gradient norm does alter
Nadam (Adam w/ Nesterov)	ARMA1, transient compensation	yes	ARMA1, transient compensation	separate, but gradient norm does alter

Ruder, Sebastian. "An overview of gradient descent optimization algorithms." arXiv preprint arXiv:1609.04747 (2016).

GENERAL OPTIMIZER STRUCTURE + SGD

parameter update: $\theta[i] = \theta[i-1] + \Delta[i]$

i ~ indexes parameter updates (*i.e.*, mini-batch)

input step/gradient (update): $\nabla[i] = \frac{\partial C}{\partial \theta[i-1]}$ $\quad g[i] = -\eta \frac{\partial C}{\partial \theta[i-1]}$

SGD:

$\Delta[i] = g[i]$

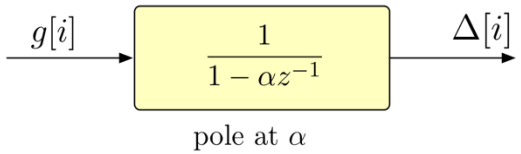
SGD with momentum:

$v[i] = \alpha v[i-1] + g[i]$

$\Delta[i] = v[i]$

v is called the "*velocity*"

α is called "*momentum*" ($\alpha \sim 0.9$)



Momentum: low-pass filter on gradient – removes high-frequency gradient noise

GRADIENT NORMALIZATION

Idea: estimate gradient RMS and normalize

parameter update: $\theta[i] = \theta[i-1] + \Delta[i]$

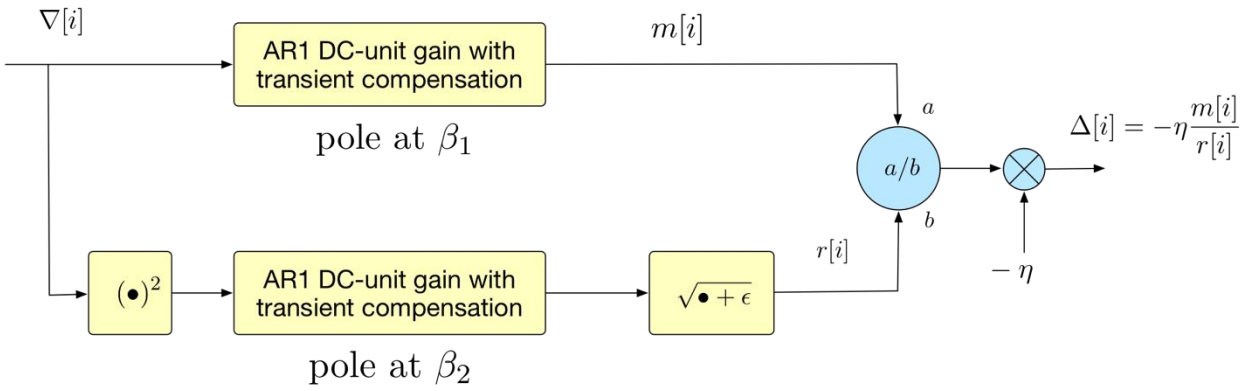
input step/gradient (update): $\nabla[i] = \frac{\partial C}{\partial \theta[i-1]}$ $g[i] = -\eta \frac{\partial C}{\partial \theta[i-1]}$

Can compute the RMS value of $\nabla[i]$ or $g[i]$

this is done by using a low-pass filter on the square of these quantities – *i.e.*, like computing the sample second moment

ADAM IN PYTORCH

https://pytorch.org/docs/stable/optim.html#torch.optim.Adam



Default:

```
def __init__(self, params, lr=1e-3, betas=(0.9, 0.999), eps=1e-8,
             weight_decay=0, amsgrad=False):
```

Tuned:

```
my_adam = optim.Adam(lr=0.002, betas=(0.92, 0.99),
eps=1e-09)
```

D. P. Kingma, K. L. Ba, ADAM: A Method for Stochastic Optimization, ICLR 2015

LEARNING RATE SCHEDULES

Change (typically decrease) the learning rate as we do more parameter updates (batches)

Recall LMS: large learning rate implies faster convergences, but more "maladjustment error" (*i.e., gradient noise*)

Could also use a LR schedule to try to force the optimizer out of a local minimum

(to go to a better local minimum, likely)

LEARNING RATE SCHEDULES IN PYTORCH

https://pytorch.org/docs/stable/optim.html#how-to-adjust-learning-rate

```
learning_rate = 0.1

optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate, momentum=0.9, nesterov=True)

# step_size: at how many multiples of epoch you decay
# step_size = 1, after every 1 epoch, new_lr = lr*gamma
# step_size = 2, after every 2 epoch, new_lr = lr*gamma

# gamma = decaying factor
scheduler = StepLR(optimizer, step_size=1, gamma=0.1)

for epoch in range(num_epochs):
    [...]
    # Decay Learning Rate
    scheduler.step()
    # Print Learning Rate
    print('Epoch:', epoch,'LR:', scheduler.get_lr())
```

Rule: apply learning rate **scheduling** **AFTER optimizer** update

```
>>> scheduler = ...
>>> for epoch in range(100):
>>>     train(...)
>>>     validate(...)
>>>     scheduler.step()
```

From LMS, we know that large learning rate implies faster convergences, but more "maladjustment error" (*i.e.*, gradient noise)

BATCH NORMALIZATION LAYER

learn the best “level" for internal activations

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

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

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

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

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

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

γ and β are trainable parameters

this normalization is done for each mini-batch but what to do when using trained network for inference?

During inference, replace the mini-batch data-average mean and variance by the data-average mean and variance over the entire dataset

11: In $N^{\mathrm{inf}}_{\mathrm{BN}}$, replace the transform $y = \mathrm{BN}_{\gamma,\beta}(x)$ with $y = \frac{\gamma}{\sqrt{\mathrm{Var}[x]+\epsilon}} \cdot x + \left(\beta - \frac{\gamma\,\mathrm{E}[x]}{\sqrt{\mathrm{Var}[x]+\epsilon}}\right)$

commonly used and effective technique in deep CNNs

Ioffe, Sergey, and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift." arXiv preprint arXiv:1502.03167 (2015).

FOLLOW HIGH-LEVEL GUIDELINES

Loss Function

Binary Classification → Use sigmoid output activation with Binary Cross Entropy Loss

M-ary Classification → Use softmax output activation with Multi-Class Cross Entropy Loss

Regression → Use linear output activation with MSE loss (L2)

Regularization → Use dropout and L2 regularization

Target network size so that:

dropout rate ~ 0.2, L2-reg coefficient ~ 1e-4

Optimizer → Adam with defaults is a good start

`optim.lr_scheduler.ReduceLROnPlateau()` or simple step LR schedules

A lot of focus on this in the literature, but designing your dataset is more important (consider above fine tuning in practice)

THIS IS HOPELESSLY COMPLEX!?!?!

We need to search over:

1. Model Architecture
    1. Number of layers
    2. Layer types
    3. Number of nodes in each layer
2. Loss Functions
3. Regularization Methods
    1. L1, L2, L1_L2
    2. Vary with layer
    3. Weight vs bias
4. Optimizers
    1. Type: SGD, Adam, etc.
    2. Parameters
    3. Learning rate schedules