Deep learning in more details

Hung-Hsuan Chen

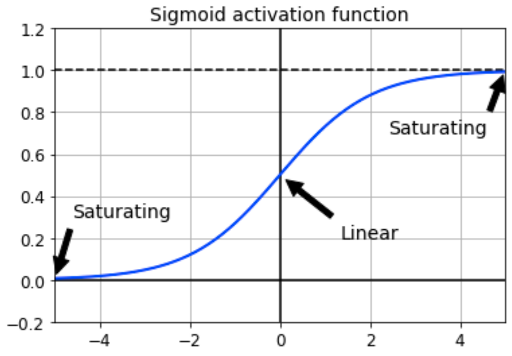
Many are adopted from Aurelien Geron's book

A summary of key component selections in DNN

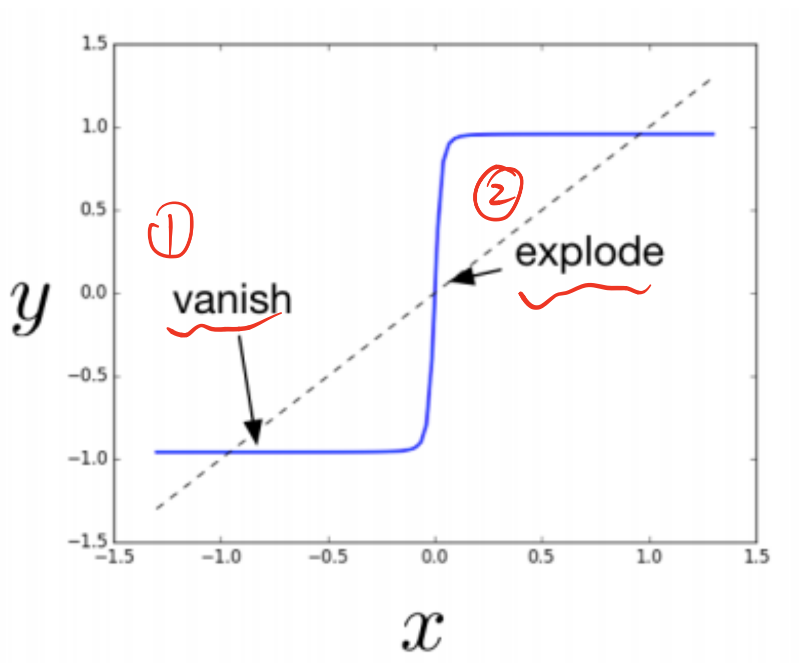
Task	# input neurons	# output neurons	Hidden activation	Output activation	Loss
Regression	# features	1	• ReLU • Leaky-ReLU • RReLU • PReLU • ELU • SELU	• None • ReLU/softplus (if positive outputs) • Logistic/tanh (if bounded outputs)	• MSE • MAE • Huber
Binary classification		1		• Logistic	• Cross entropy
Multi-class classification		# classes		• Softmax	
Multi-label binary classification		# labels		• Logistic	

The vanishing gradient problem

- Vanishing gradient problem in sigmoid and tanh
  - When z is outside $[-4, 4]$, $f'_{sigmoid}(z) \approx 0$ and $f'_{tanh}(z) \approx 0$ ➔ new information cannot be back-propagated



The exploding gradient problem



Xavier (a.k.a., Glorot) initialization

- Xavier Glorot and Yosha Bengio discussed the vanishing/exploding gradient issue in [1]
- Xavier initialization: randomly initialize the link weights by one of the following (useful when using logistic activation)
  - $w \sim N(\mu = 0, \sigma^2 = 1/fan_{avg})$
    - $fan_{avg} = \frac{1}{2}(fan_{in} + fan_{out})$
      - fan_{in} and fan_{out} are the # inputs and # neurons of a layer
  - $w \sim Unif(-r, r)$
    - $r = \sqrt{\frac{3}{fan_{avg}}}$
- Such initialization may accelerate the training of deep networks

[1] Xavier Glorot and Yosha Bengio, “Understanding the difficulty of training deep feedforward neural networks”, AIStat 2001.

Intuition of why Xavier initializer works

- If $z_i = w_0 + w_1 x_{i1} + \cdots + w_d x_{id}$, we don't want $\sigma(z)$ getting "saturated", i.e., $w_0 + w_1 x_{i1} + \cdots + w_d x_{id}$ should not be too large or too small
- For $x_{i1}, \ldots, x_{id}$, we can scale or normalize them (e.g., $x_{ij} = \frac{x_{ij} - \overline{x_{*j}}}{\mathrm{std}(x_{*j})}$) such that $E[x_{*j}] = 0, Var(x_{*j}) = 1$
- If each $w_j x_{ij}$ is uncorrelated and has the same variance, then
$$Var(z_i) = Var(w_0 + w_1 x_{i1} + \cdots + w_d x_{id}) \approx dVar(w_j x_{ij})$$
- Assuming $E[w_j] = E[x_{ij}] = 0$:
$$dVar(w_j x_{ij}) = d\left[(E[w_j])^2 Var(x_{ij}) + (E[x_{ij}])^2 Var(w_j) + Var(w_j)Var(x_{ij})\right]$$
$$= d[Var(w_j)Var(x_{ij})] = dVar(w_j)$$
- In order to make $Var(z_i) \approx 1$, each $Var(w_j) \approx \frac{1}{d}$
- Likewise, for "backprop" signal, $Var(w_j) \approx \frac{1}{n_{out}}$
- As a compromise, set $w \sim N(\mu = 0, \sigma^2 = \frac{1}{0.5(fan_{in} + fan_{out})})$

Famous weight initializers

Name	Activation function	Initialized by random normal	Initialized by random uniform
Xavier/ Glorot	Logistic, softmax,	$N\left(\mu = 0, \sigma^2 = \frac{1}{fan_{avg}}\right)$	$U(-r, +r), r = \sqrt{\frac{3}{fan_{avg}}}$
	tanh	$N\left(\mu = 0, \sigma^2 = \frac{16}{fan_{avg}}\right)$	$U(-r, +r), r = 4\sqrt{\frac{3}{fan_{avg}}}$
He	ReLU, ELU, and variants	$N\left(\mu = 0, \sigma^2 = \frac{2}{fan_{in}}\right)$	$U(-r, +r), r = \sqrt{\frac{6}{fan_{in}}}$
Lecun	Logistic, softmax, tanh	$N\left(\mu = 0, \sigma^2 = \frac{1}{fan_{in}}\right)$	$U(-r, +r), r = \sqrt{\frac{3}{fan_{in}}}$

the number of input neuron

input+ output

2

Non-saturating activation functions

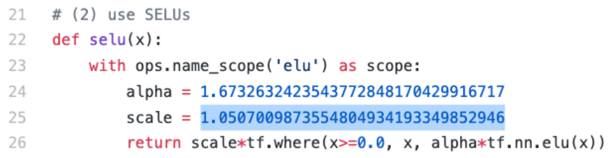
- Rectified Linear Unit (ReLU)
- LeakyReLU
- Randomlized leaky ReLU (RReLU)
- Parametric Leaky ReLU (PReLU)
- Exponential Linear Unit (ELU)
- Scaled ELU (SELU)

Non-saturating activation functions

- ReLU: $f(z) = \max(z, 0) = \begin{cases} z, & z \geq 0 \\ 0, & z < 0 \end{cases}$
  - It does not saturate for positive values
  - Problem: dying ReLU, a neuron "dies" when the weighted sum of the inputs are negative for all instances
    - Although sometimes a dead neuron may "awake", it takes a long time
- Leaky ReLU: $f_\alpha(z) = \max(z, \alpha z)$
  - A neuron may not really "dead" because the gradient is at least α

Non-saturating activation functions

- RReLU: $f(z) = \max(z, \alpha z)$
  - $\alpha \sim Unif(\ell, u)$ during training
  - $\alpha = \frac{\ell+u}{2}$ (averaged) during testing
- PReLu: $f(z) = \max(z, \alpha z)$
  - α is learned via backprop
- ELU: $f_\alpha(z) = \begin{cases} z, & z \geq 0 \\ \alpha(e^z - 1), & z < 0 \end{cases}$
- SELU: $f_\alpha(z) = \lambda \begin{cases} z, & z \geq 0 \\ \alpha(e^z - 1), & z < 0 \end{cases}$, where
  - $\alpha = 1.6732632423543772848170429916717$
  - $\lambda = 1.0507009873554804934193349852946$



```
# (2) use SELUs
def selu(x):
    with ops.name_scope('elu') as scope:
        alpha = 1.6732632423543772848170429916717
        scale = 1.0507009873554804934193349852946
        return scale*tf.where(x>=0.0, x, alpha*tf.nn.elu(x))
```

Source of SELU: https://github.com/bioinf-jku/SNNs/blob/master/selu.py

Ref of SELU paper: https://arxiv.org/pdf/1706.02515.pdf

When to use which activation function (in the hidden layer)?

- ReLU is still popular, probably because many libraries provide ReLU-specific optimizations
- In general, SELU > ELU > (Variants of) Leaky ReLU > ReLU > tanh > logistic
  - But may still need experiments to decide
- If the network architecture prevents it from self-normalizing, ELU > SELU
- If run-time latency is crucial, use LeayReLU
- If having huge training data, try PReLU

Batch normalization (1/2)

- Adding operations before and after activation function (in hidden layer)
- For every batch:
  - Zero-center and normalize inputs
  - Scale and shift (so that mean and standard deviation can beyond 0 and 1)

- Batch norm training

1. $\boldsymbol{\mu}_B = \frac{1}{m_B}\sum_{i=1}^{m_B} \boldsymbol{x}^{(i)}$
2. $\boldsymbol{\sigma}_B^2 = \frac{1}{m_B}\left(\boldsymbol{x}^{(i)} - \boldsymbol{\mu}_B\right)^2$
3. $\widehat{\boldsymbol{x}}^{(i)} = \frac{\boldsymbol{x}^{(i)} - \boldsymbol{\mu}_B}{\sqrt{\boldsymbol{\sigma}_B^2 + \epsilon}}$
4. $\boldsymbol{z}^{(i)} = \boldsymbol{\gamma} \odot \widehat{\boldsymbol{x}}^{(i)} + \boldsymbol{\beta}$
   - $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ are learned by backprop
   - $\boldsymbol{\mu}_B$ and $\boldsymbol{\sigma}_B$ are estimated during training

Batch normalization (2/2)

↳ 对於多層 很 important

- Batch norm testing
  - Given a new test instance, how to perform batch norm?
    - There is no “batch”, so how to compute $\boldsymbol{\mu}_B$ and $\boldsymbol{\sigma}_B^2$?
- Estimate $\boldsymbol{\mu}_B$ and $\boldsymbol{\sigma}_B^2$ by visit entire training dataset again?
  - Usually too costly
- Estimate $\boldsymbol{\mu}_B$ and $\boldsymbol{\sigma}_B^2$ by moving average
  - At training step t

$$\hat{\boldsymbol{\mu}}_B[t] = \underbrace{\hat{\boldsymbol{\mu}}_B}_{\text{previous}}[t-1]\times\text{momentum} + \underbrace{\boldsymbol{\mu}_B}_{\text{current}}[t]\times(1-\text{momentum})$$
$$\hat{\boldsymbol{\sigma}}_B^2[t] = \hat{\boldsymbol{\sigma}}_B^2[t-1]\times\text{momentum} + \boldsymbol{\sigma}_B^2[t]\times(1-\text{momentum})$$

  - Momentum, a.k.a., decay, is usually set to 0.9

previous

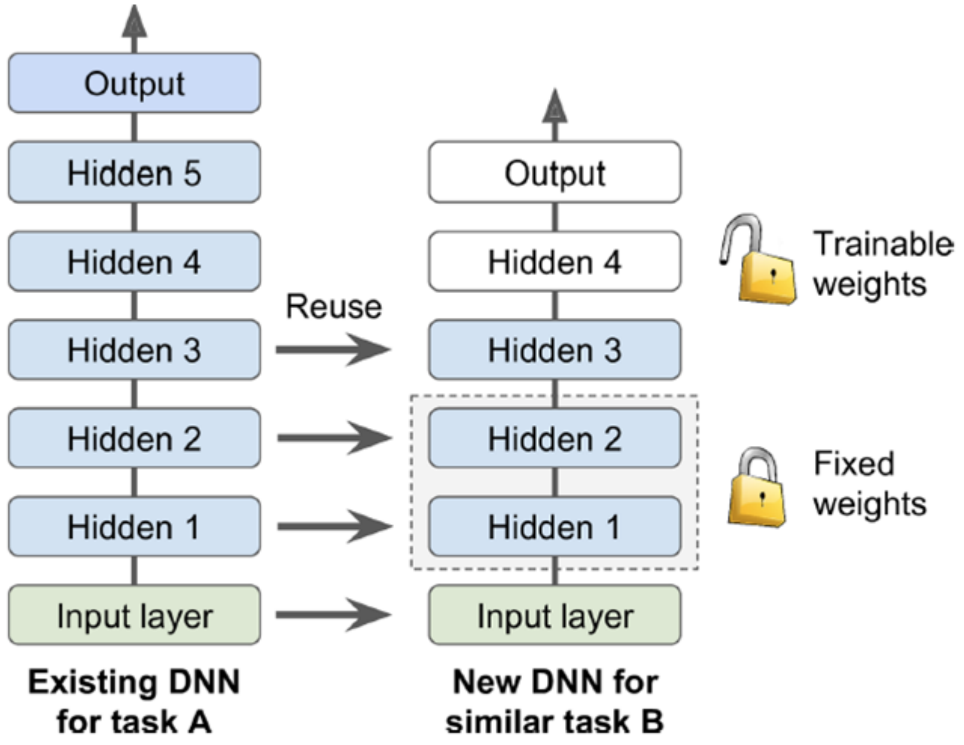
current

Gradient clipping

↳ prevent gradient explore

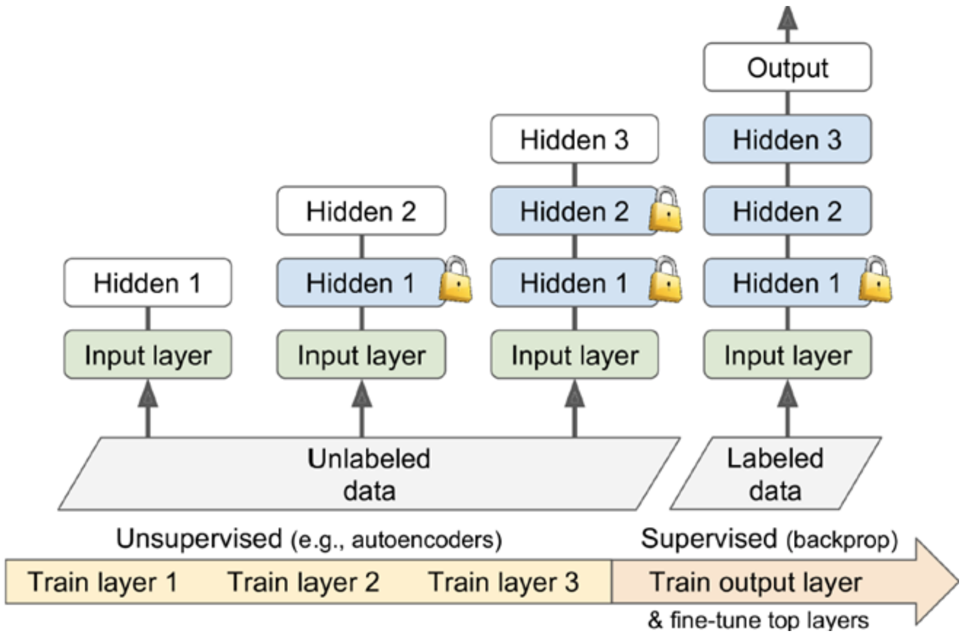
- For RNN, it could be tricky to apply Batch norm
- Manually clip the gradient when it is too large or too small
- Example: set upper bound to 1, lower bound to -1
  - If gradient is larger than 1, output 1
  - If gradient is smaller than -1, output -1

Transfer learning via pretrained layers



Layer-wise pretraining

- Used to be a popular method in ~2010
- May still be useful when # training instances is limited



Various optimizers

- SGD
- Momentum
- Nesterov Accelerated Gradient
- AdaGrad
- RMSProp
- Adam
- Nadam

SGD

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$$

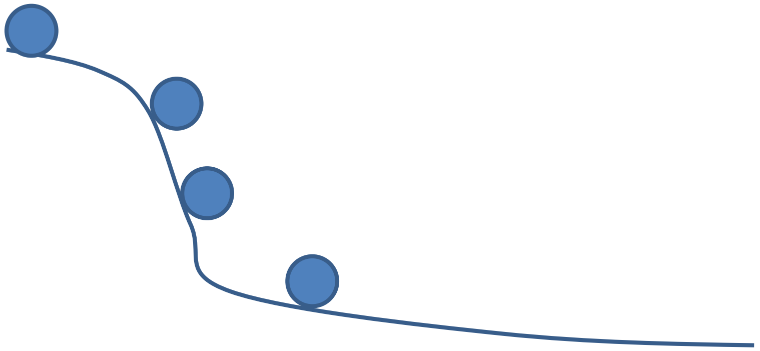
- η: learning rate, usually a small positive value, e.g., 0.001

- A parameter's next value is the current value shifted toward the opposite direction of the gradient

Momentum

$$\boldsymbol{m} \leftarrow \beta \boldsymbol{m} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$$
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{m}$$

- β: momentum, usually set to 0.9
  - Large β: no friction
  - Small β: high friction
  - $\beta = 0$: SGD



Nesterov Accelerated Gradient (NAG)

- Look ahead: measure the gradient of $J(\boldsymbol{\theta})$ slight ahead the current $\boldsymbol{\theta}$
  - It works probably because the momentum vector usually points to the right direction

$$\boldsymbol{m} \leftarrow \beta\boldsymbol{m} - \eta\nabla_{\boldsymbol{\theta}}J(\boldsymbol{\theta} + \beta\boldsymbol{m})$$
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{m}$$

- β: momentum, usually set to 0.9

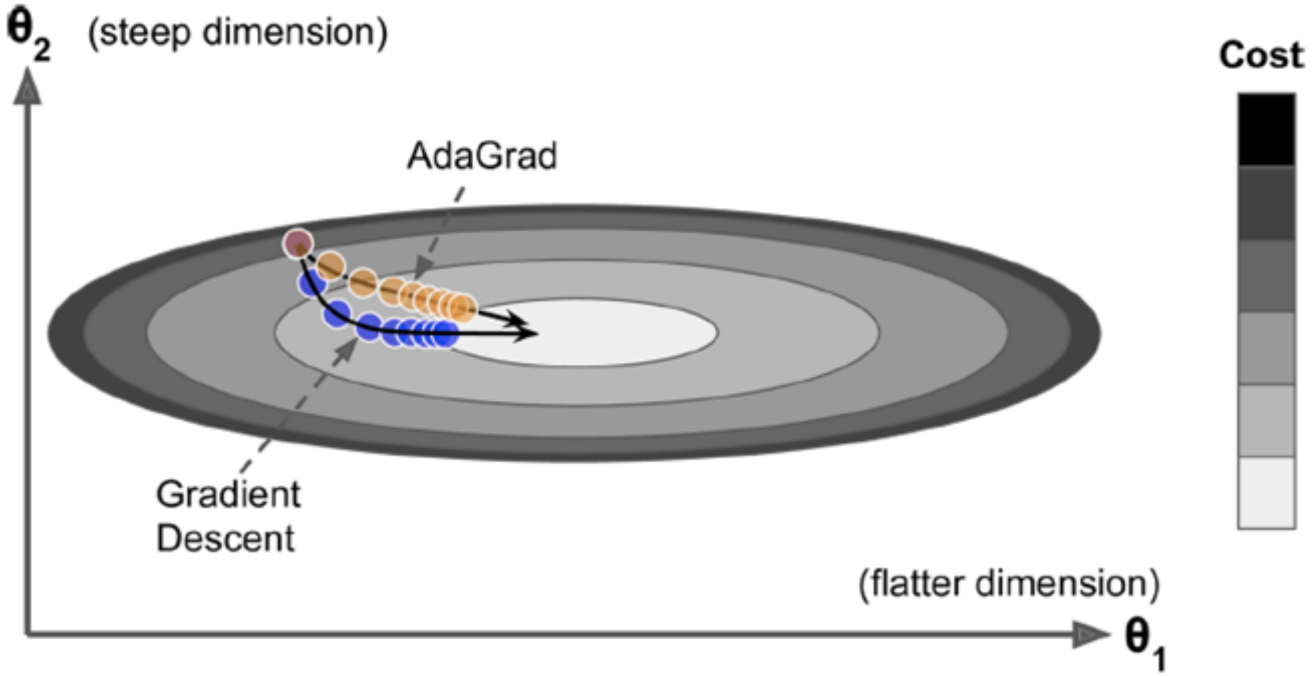
AdaGrad (1/2)

$$\boldsymbol{s} \leftarrow \boldsymbol{s} + \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \odot \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \frac{\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})}{\sqrt{\boldsymbol{s} + \epsilon}}$$

  - ϵ: smoothing term to avoid divide-by-zero, usually set to 10^{-9}
- If $\partial J(\boldsymbol{\theta}) / \partial \theta_i$ is large for certain i, square the value will make the the corresponding S_i larger
- AdaGrad decays the learning rate faster for the steep dimensions and gentler for the others
- In practice, AdaGrad usually works well on shallow networks, but less satisfactory on deep networks
  - Probably because learning rate is scaled down too much

AdaGrad (2/2)


θ_2 (steep dimension)
AdaGrad
Gradient Descent
Cost
(flatter dimension)
θ_1

RMSProp

- Instead of accumulating all previous gradients to decay the learning rate (as in Adagrad), RMSProp only accumulates recent gradients

$$\boldsymbol{s} \leftarrow \beta \boldsymbol{s} + (1-\beta)\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \odot \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$$
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \frac{\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})}{\sqrt{\boldsymbol{s} + \epsilon}}$$

  - β: decay rate, usually set to 0.9

Adaptive moment estimation (Adam)

- Combines momentum and RMSProp
  - Track exponentially decaying average gradients
  - Track exponentially decaying average squared gradients

1. $\boldsymbol{m} \leftarrow \beta_1 \boldsymbol{m} - (1 - \beta_1)\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
2. $\boldsymbol{s} \leftarrow \beta_2 \boldsymbol{s} + (1 - \beta)\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \odot \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
3. $\widehat{\boldsymbol{m}} \leftarrow \frac{\boldsymbol{m}}{1-\beta_1^t}$
4. $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1-\beta_2^t}$
5. $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \frac{\widehat{\boldsymbol{m}}}{\sqrt{\hat{\boldsymbol{s}}+\epsilon}}$
   - t: iteration count
   - Step 3 and 4 are used to boost the value of $\boldsymbol{m}$ and $\boldsymbol{s}$ when t is small
   - β_1 and β_2 are usually set to 0.9 and 0.999

NADAM

- Adam optimization + Nesterov “look-ahead” trick (i.e., measuring the gradient of $J(\boldsymbol{\theta})$ slight ahead the current $\boldsymbol{\theta}$)

Summary of optimizer selection

- Adaptive optimization methods (e.g., RMSProp, Adam, Nadam) usually perform well
- However, study also showed that occasionally SGD or Nesterov Accelerated Gradient perform better
- Probably try adaptive optimization first
- How about Hessian (i.e., second-order partial derivatives) methods?
  - Requires n^2 computation time (n: # parameters)
  - Too costly for deep learning

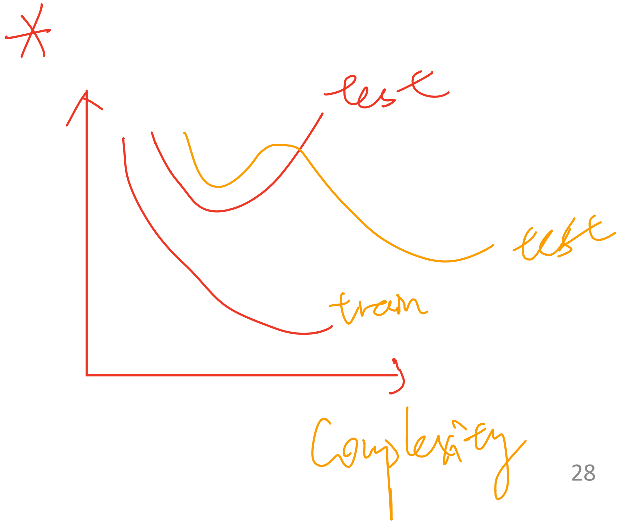
Learning rate scheduling

- Exponential scheduling: $\eta(t) = \eta_0 r^{t/s}$
  - t: iteration count; s:step number; r: decay rate
- Power scheduling: $\eta(t) = \eta_0/(1 + t/s)^c$
- Piecewise constant scheduling: use a constant learning rate for some epochs, and a smaller rate for another number of epochs, and so on
- Performance scheduling: measure the validation error every c steps and reduce the learning rate when error stops dropping

Regularization techniques

$\hookrightarrow$ prevent overfitting

- Weight penalty
- Data augmentation
- Early stopping
- Dropout



Regularization – weight penalty

- Similar to what we've used in linear regression and logistic regression
- L2-norm
  - Penalize the squared values of the weights
  - Tend to shrink large weights to smaller values
- L1-norm
  - Penalize the absolute values of the weights
  - Tend to shrink some weights to zero; while other weights may still be large
- A combination of both (e.g., elastic-net)

Regularization – data augmentation

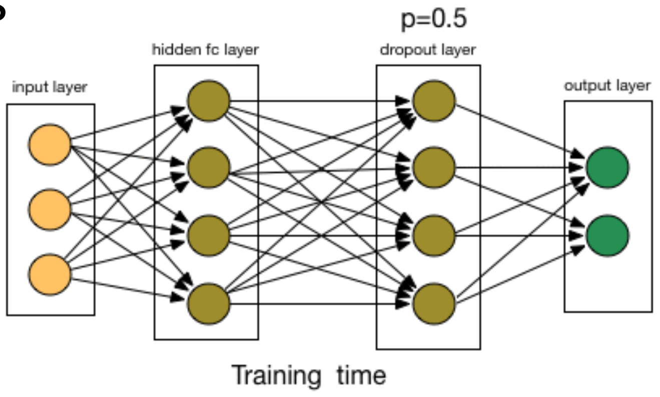
- More data leads to less variance
  - Less likely to overfitting
- Collect more data
- Add synthetic data
  - E.g., to increase the synthetic training data for object recognition from images, popular techniques include
    - Rotate images
    - Scale image
    - Translating a few pixels
    - Injecting random negatives pictures
  - Seems no principle way to generate synthetic data across different domains

Regularization – early stopping

- Training the model based on training data
- Validate the performance of the current model based on validation data
  - Assume that validation data are representative of future (unseen) data
- Once the validation performance starts get worse, stop training
  - Although training performance continues improving, the model may starts learning the “noise” in the training data

Regularization – dropout

- Randomly remove some nodes and their incoming and outgoing connections
- If randomly remove certain neurons the model can still learn something, the model is probably more robust and more resilient to noise



Summary

- When networks become “deep”, simple SGD may be problematic, because of vanishing and exploding gradient
- Many techniques were proposed to (partially) solve the problem to accelerate learning
  - Weight initialization
  - Activation function design
  - Optimizer design
  - …