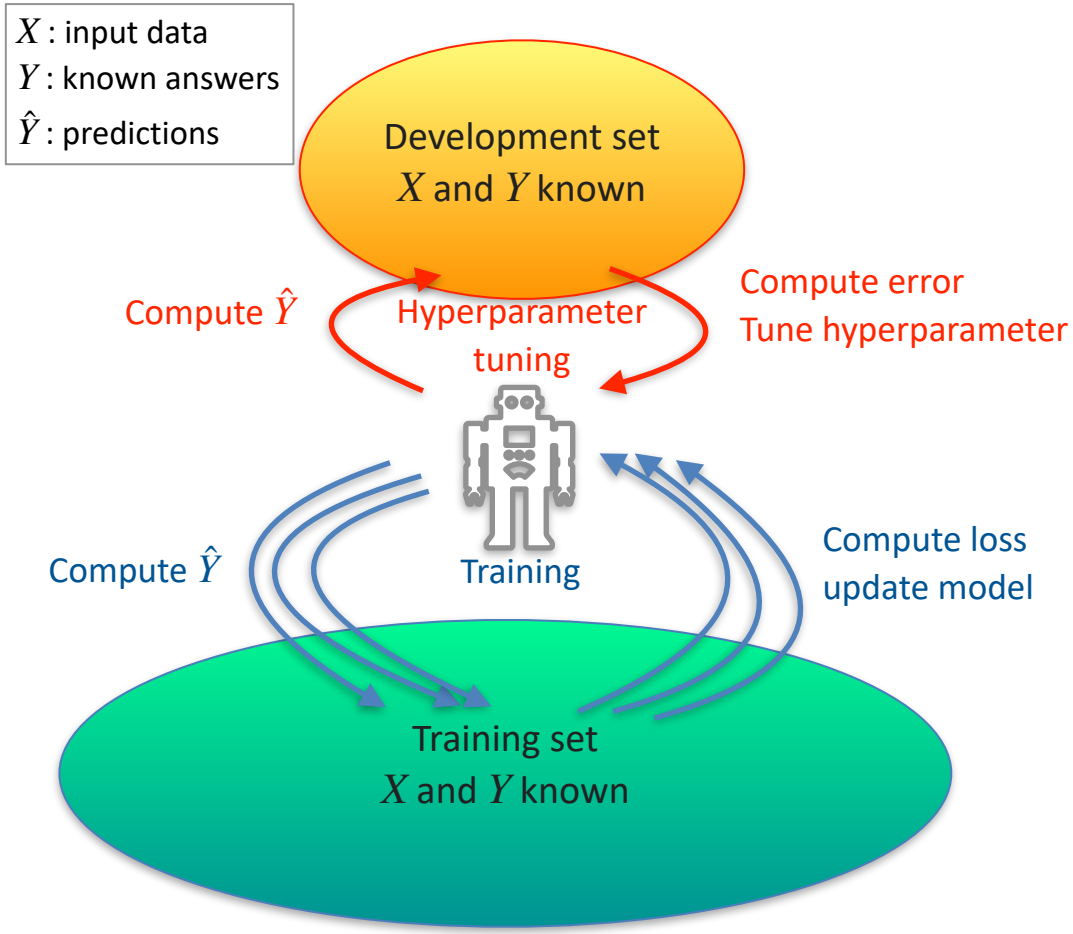
Optimizing Deep Learning Models

Bias-Variance Tradeoff, L_2 Regularization and Dropouts, Vanishing Gradient, Mini-batch Gradient Descent, RMSProp, Adam Optimizer

Debapriyo Majumdar

debapriyo@isical.ac.in

Training, Development and Test Sets



- Fix a function or architecture
- Training: train (iteratively) the parameters (weights W and bias b for all layers) using training data
- Once the model converges, validate results with development set (dev set)
- Tune hyperparameters (number of layers, sizes, or even the architecture) and train again
- *Often, we say test set and mean dev set instead*
- Test set: data for which the known answers are never used to influence the model (by direct training or hyperparameter tuning)

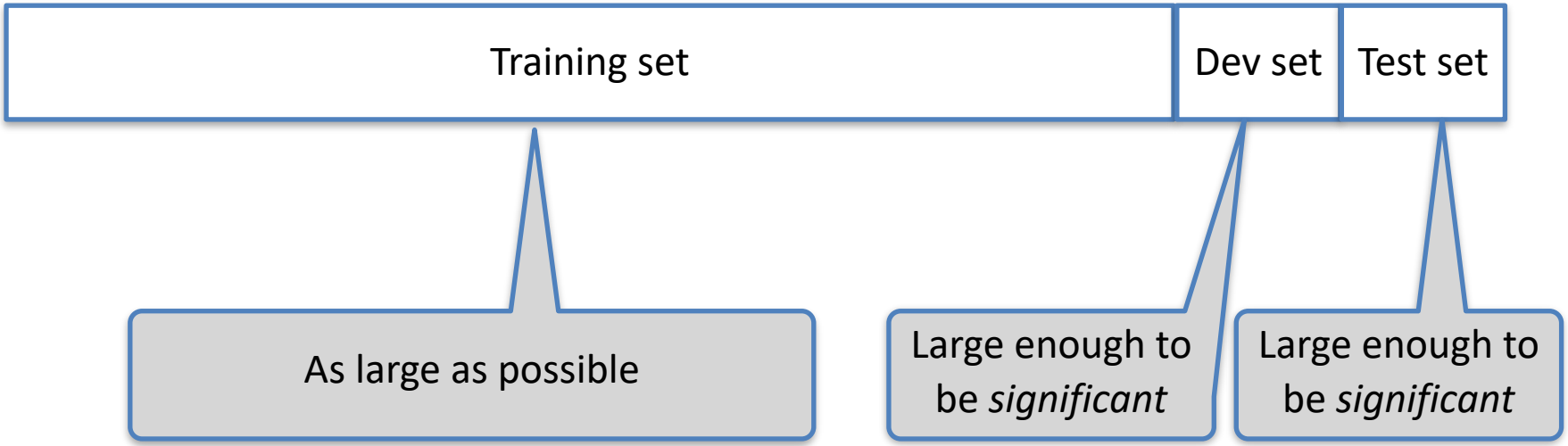
Training, Dev and Test Sets

- Dev set: for hyperparamter tuning, cross-validation
- Test set: final performance measure, should not be used for tuning
- Important: dev set and test set should be from the same distribution
- Training a model on the training set (known input)
  - Minimize training error $\approx$ cost function
- Actual goal: the model needs to perform well on the test set (unobserved input)
- Generalization: ability to perform well on previously unobserved input

Training, Dev and Test Sets

Old thumb-rule

If you have a lot of data



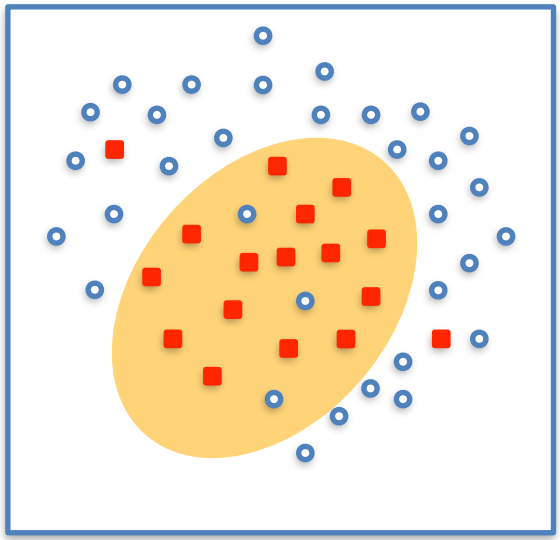
Bias and Variance

- Let $x_1, \cdots, x_m$ be a set of data points and $y_1, \cdots, y_m$ be known values associated
- Assumption: there is a function f such that $f(x) = y + \varepsilon$, where ε is the *noise* with mean 0 and variance σ^2
- What we do: we model f by a function $\hat{f}$

- Bias is the expected error $E[y - \hat{f}(x)]$ of our prediction
- Variance is the variance of the predictions: $\mathrm{Var}(\hat{f}(x)) = E[\hat{f}(x) - E(\hat{f}(x)]$
  - Intuitively: for small variations of x, how much $\hat{f}(x)$ varies

The Bias-Variance Tradeoff

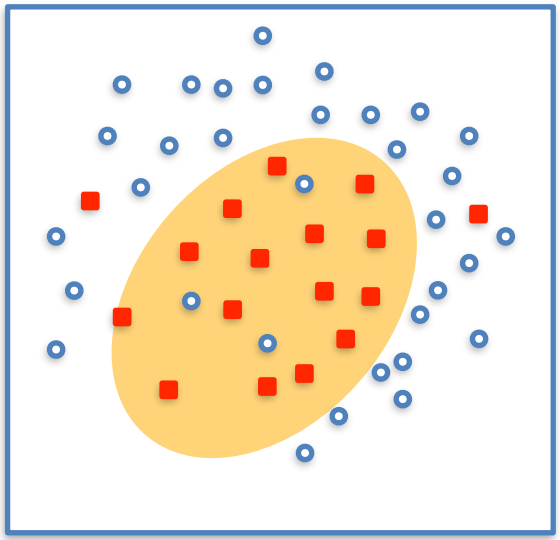
Training set



Training error: **low**

Intuitive
Correct
Classification

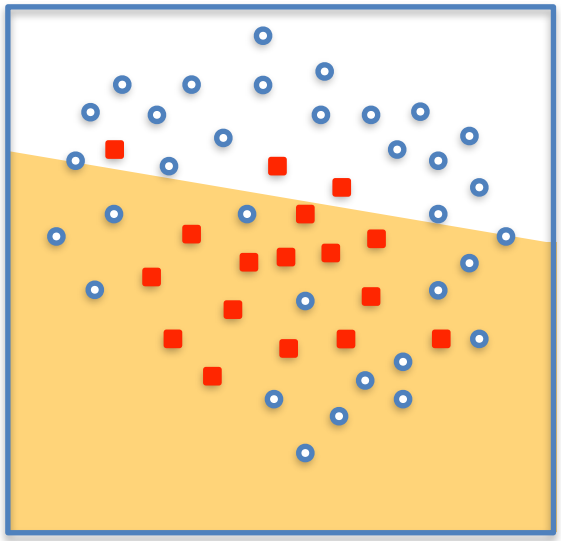
Test set



Test error: **low**

The Bias-Variance Tradeoff

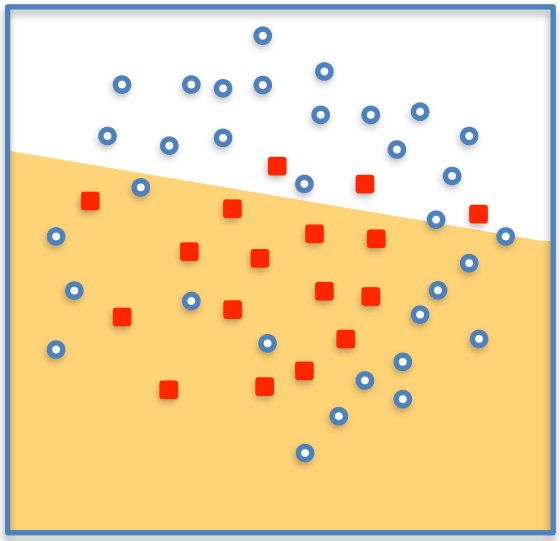
Training set



A simple model

Underfitting

Test set



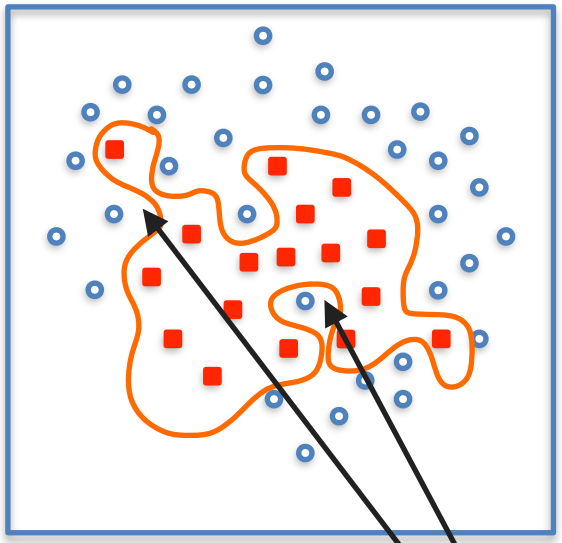
Training error: **high**

Test error: **high**

- High Bias
- Low variance: if x changes slightly, $\hat{f}(x)$ does not change

The Bias-Variance Tradeoff

Training set

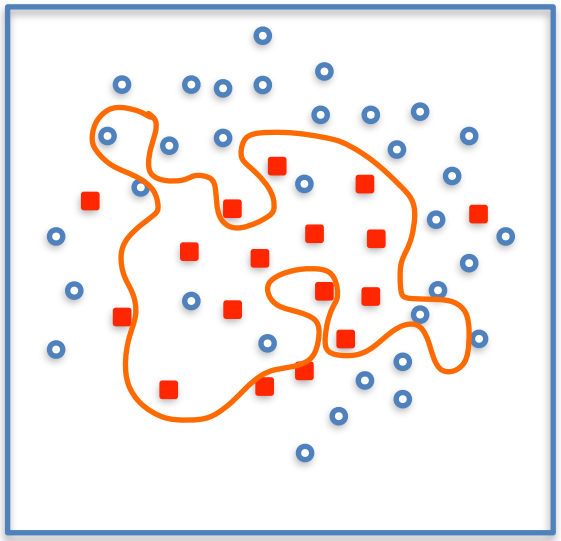


Training error: **low**

A model with much higher capacity

Overfitting

Test set



Test error: **relatively high**

- Low Bias
- High variance: if x changes slightly, $\hat{f}(x)$ changes in many regions

The Bias-Variance Tradeoff

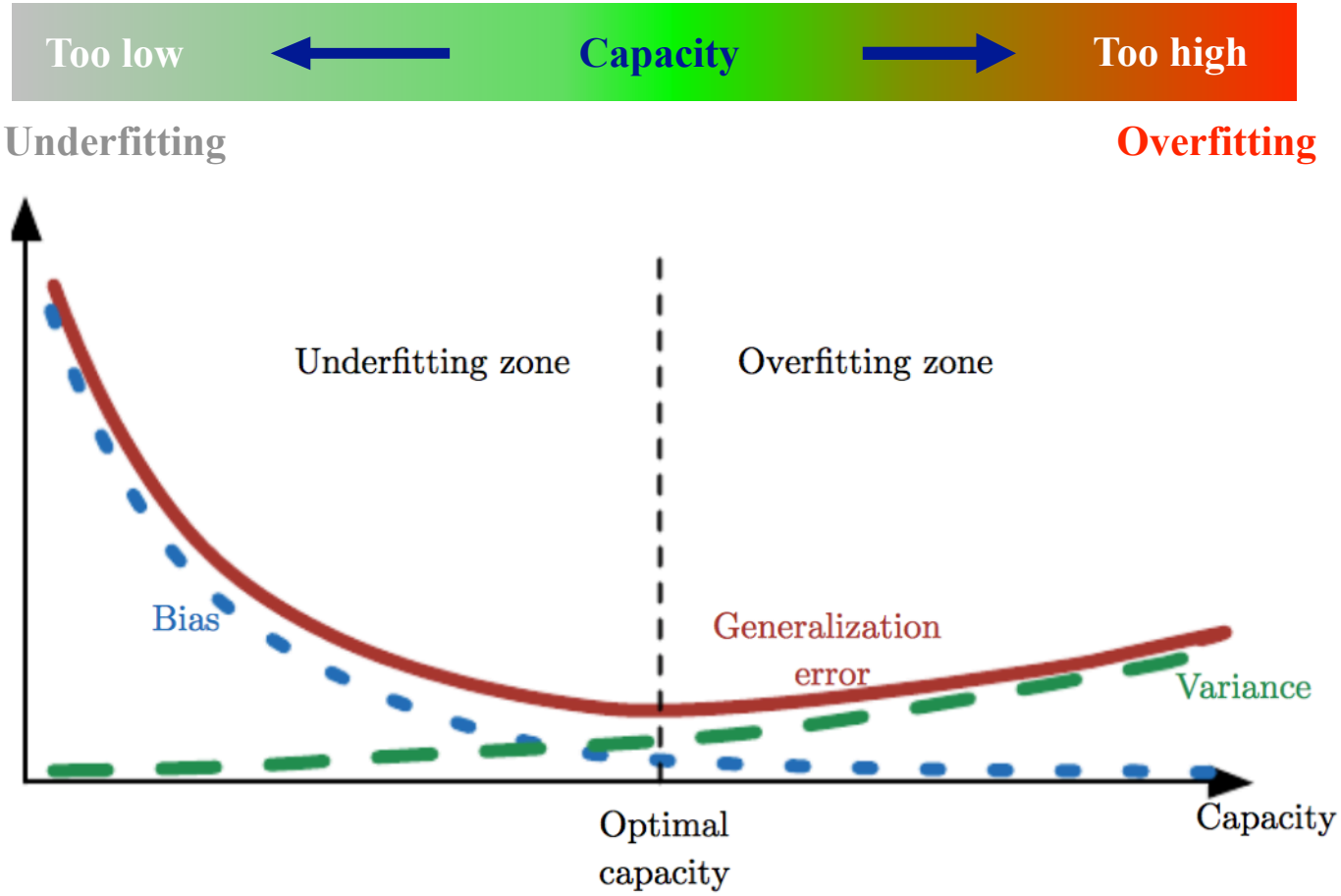


Figure courtesy: www.deeplearningbook.org

The Bias-Variance Tradeoff

Model too simple
More layers, change architecture

Overfitting
Get more training data, use regularization

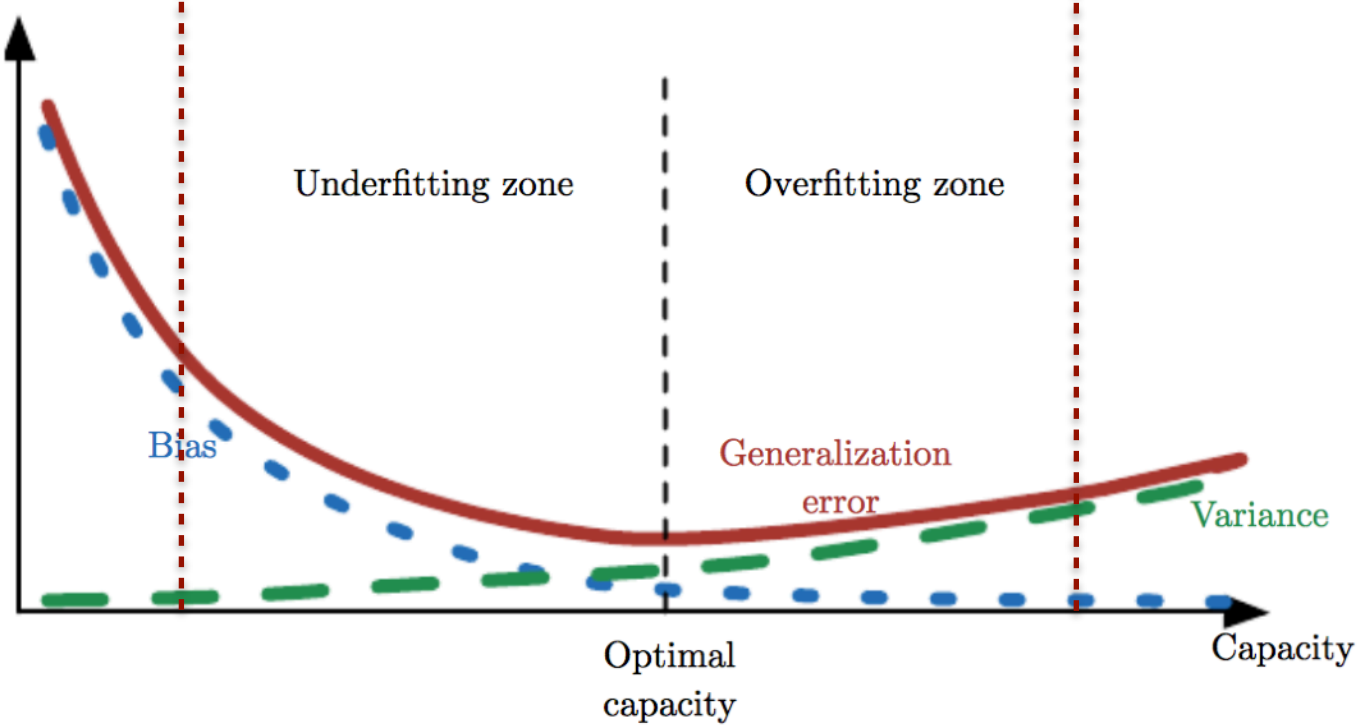


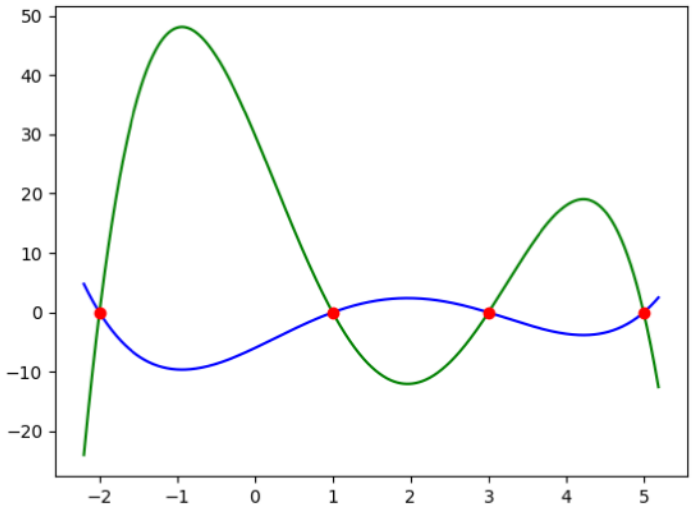
Figure courtesy: www.deeplearningbook.org

Regularization: intuition using linear regression

- Optimization function (mean squared error):

$$J(\boldsymbol{w}) = \mathrm{MSE}_{\mathrm{train}} + \lambda \boldsymbol{w}^T \boldsymbol{w}$$

- Higher values of weights $\boldsymbol{w}$ increases the cost function, so the algorithm tries to keep the weights small
- Smoothening effect
- Prevents overfitting
- Parameter λ:
  - Very high: too much regularization, all weights close to zero
  - Very small: no regularization



$$y = -x^4 + 7x^3 - 5x^2 - 31x + 30$$

$$y = \frac{x^4}{5} - \frac{7x^3}{5} + x^2 + \frac{31x}{5} - 6$$

Picture source:https://www.datacamp.com/community/tutorials/towards-preventing-overfitting-regularization

L2 (alternatively L1) Regularization

- Approach: add a fraction of the weights to the cost function
- The optimization function (loss function that we need to **minimize**):

$$J(\theta) = \frac{1}{m} L(W^{[1]}, b^{[1]}, \cdots, W^{[l]}, b^{[l]}) + \frac{\lambda}{2m} \sum_{k=1}^{l} \| W^{[k]} \|_2^2 + \frac{\lambda}{2m} \sum_{k=1}^{l} \| b^{[k]} \|_2^2$$

λ : regularization parameter

Frobenius norms of all the weight matrices

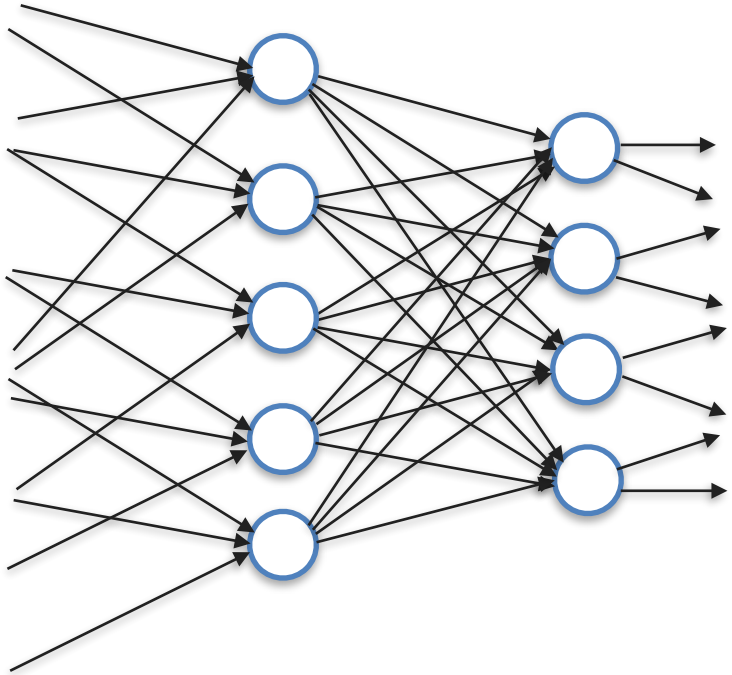
L_2 norms of all the bias vectors (matters less)

- The gradient calculation: $\frac{\partial J}{\partial W^{[k]}} = \frac{\partial L}{\partial W^{[k]}} + \frac{\lambda}{m} W^{[k]}$

Weight decay

- The parameter update: $W^{[k]} := W^{[k]} - \alpha \left(\frac{\partial L}{\partial W^{[k]}} + \frac{\lambda}{m} W^{[k]} \right) = W^{[k]} - \frac{\alpha \lambda}{m} - \alpha \frac{\partial L}{\partial W^{[k]}}$

Regularization using Dropout



- Approach: Fix a probability p
  - Usually between 0.6 and 0.8, but can be as aggressive as 0.5 too
- At every iteration of training (forward and corresponding backward propagation), for each node, drop it with probability $1 - p$
- Those nodes are back again in the next iteration, again drop nodes randomly
- Intuition:
  - Dropping a few nodes makes the network simpler
  - Cannot depend heavily on any single node, must distribute weights, hence reduce the norm of the weight matrix

Vanishing Gradient in Very Deep Networks

Layer k

Input layer 0

$a^{[0]} = x$

$$z^{[k]} = W^{[k]} a^{[k-1]} + b^{[k]}$$

$$a_i^{[k]} = g_k(z_i^{[k]})$$

Output layer l

$a^{[l]} = \hat{y}$

The gradient $\nabla_{a^{[k-1]}} J$ has a multiplicative factor $(W^{[k]})^T$ and $g^{[k]'}$, for all k

$$\frac{\partial L}{\partial a^{[k-1]}} = \frac{\partial L}{\partial z^{[k]}} \cdot \frac{\partial z^{[k]}}{\partial a^{[k]}}$$

$$= (W^{[k]})^T \frac{\partial L}{\partial z^{[k]}}$$

$$\frac{\partial L}{\partial z_i^{[k]}} = g^{[k]'}(z_i^{[k]}) \frac{\partial L}{\partial a_i^{[k]}}$$

element-wise

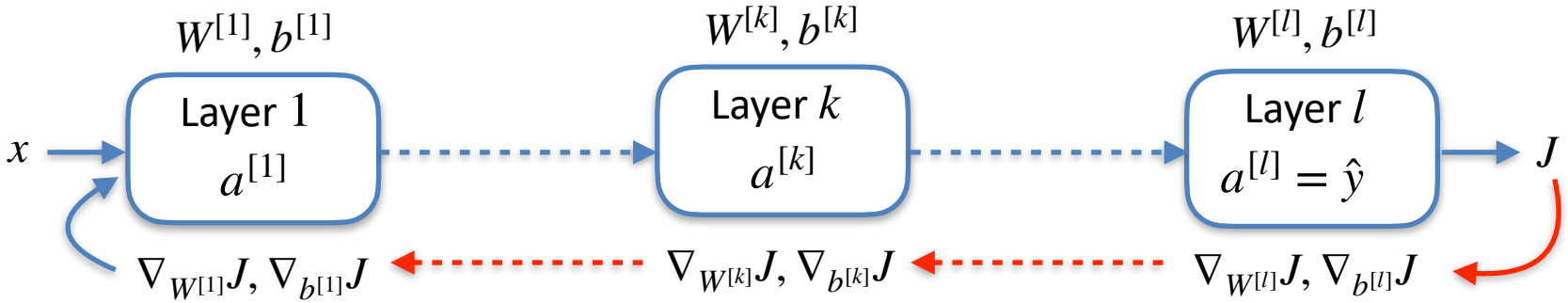
$$\frac{\partial L}{\partial W^{[k]}} = \frac{\partial L}{\partial z^{[k]}} \cdot \frac{\partial z^{[k]}}{\partial W^{[k]}} = \frac{\partial L}{\partial z^{[k]}} (a^{[k-1]})^T$$

$$\frac{\partial L}{\partial b^{[k]}} = \frac{\partial L}{\partial z^{[k]}} \cdot \frac{\partial z^{[k]}}{\partial b^{[k]}} = \frac{\partial L}{\partial z^{[k]}}$$

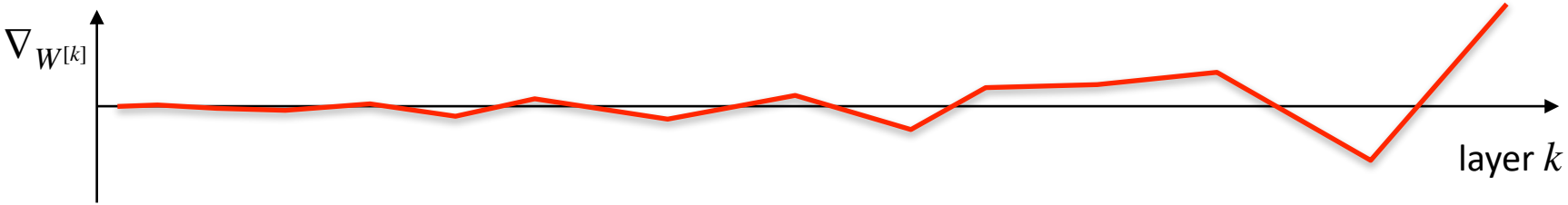
Input

$$\frac{\partial L}{\partial a^{[k]}}$$

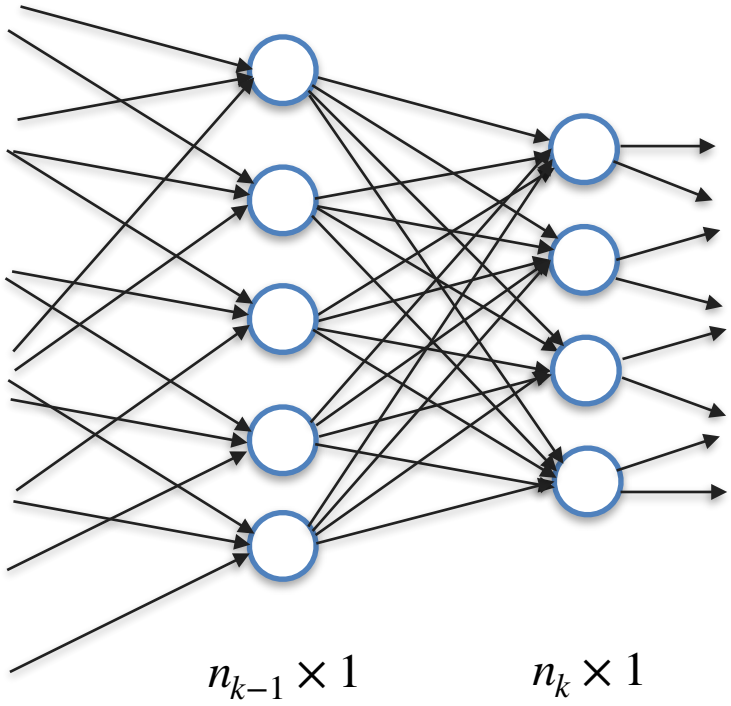
Vanishing Gradient in Very Deep Networks



- Backward propagation: gradient $\nabla_{a^{[k-1]}} J$ has a factor $(W^{[k]})^T$ and $g^{[k]'}$, for all k
- Weights $w_{ij}^{[k]}$ and derivatives $g^{[k]'} > 1$ may result in ***explosion*** of the gradients
  - Solution: gradient clipping
- Weights $w_{ij}^{[k]}$ and derivatives $g^{[k]'} < 1$ may result in the gradient being ***vanishingly small***
  - The updating of parameters become very slow, never reaches optimality



Weight Initialization



$n_{k-1} \times 1$ $\quad$ $n_k \times 1$

Goal: try so that the values $z_i^{[k]}$ do not become too much bigger or smaller than 1

- Layer k: each $z_i^{[k]}$ is a linear combination of n_{k-1} inputs to this layer
- Initialize $w_{ij}^{[k]} := \rho \cdot c$ where $\rho \sim U(0,1)$
- Many approaches for the parameter c
- For activation ReLU

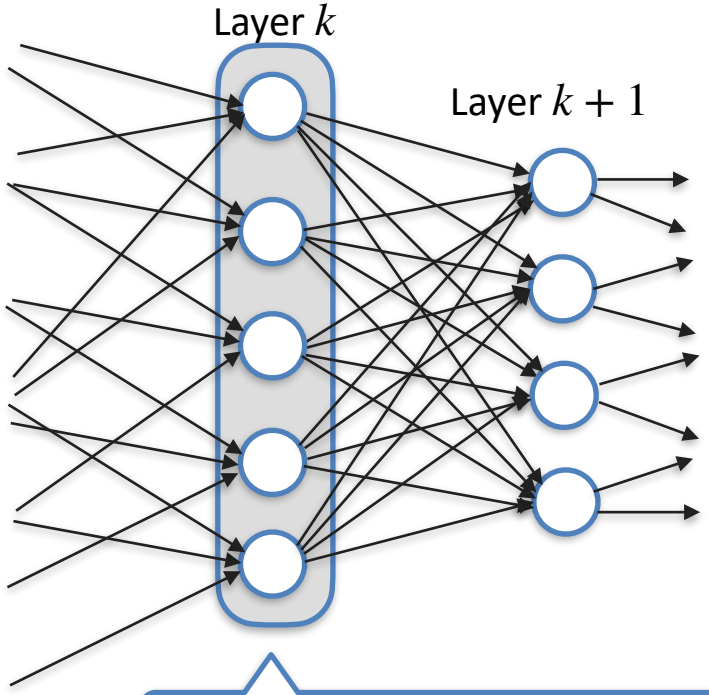
  Make the variance close to $\frac{2}{n_{k-1}}$

  $$c = \sqrt{\frac{2}{n_{k-1}}}$$
- For activation tanh

  $c = \sqrt{\frac{1}{n_{k-1}}}$, or another approach $c = \sqrt{\frac{1}{n_k + n_{k-1}}}$

Batch Normalization

Notation: $a^{[k](i)}$ = the activation vector corresponding to the i-th training example



- The values (activation) of one layer is the input to the next layer
- Centering (making mean 0) and normalizing (making variance 1) the input is a usual process
- During training, the values of layers experience *internal covariate shift* (mean and variance changes)
- Batch normalization: for each *mini-batch*, apply normalization (note: mini-batches will be explained in the later slides, for now just imagine the full data)

Center and normalize each dimension of the activation $a^{[k](i)}$ for each training example.

Ioffe, Sergey; Szegedy, Christian (2015). "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". arXiv:1502.03167

Batch Normalization

- Let $a^{[k](i)}$, or simply $a^{(i)}$ denote the activation corresponding to the i-th training example in the mini-batch B.
- Mean activation vector: $\frac{1}{m}\sum_{i=1}^{m} a^{(i)} = \mu = (\mu_1, \ldots, \mu_{n_k})$; Variance vector: $\sigma = (\sigma_1, \ldots, \sigma_{n_k})$ with

  $\sigma^2 = \frac{1}{m}\sum_{i=1}^{m} (a^{(i)} - \mu_B)^2.$
- Centering and normalization: $x_d^{(i)} = \frac{a_d^{(i)} - \mu_d}{\sqrt{\sigma_d^2 + \epsilon}}$ for each dimension $d = 1,\ldots, n_k$.
  - The small constant ϵ is added for numerical stability.
  - After centering and normalization, the resulting activations have mean 0 and variance 1 (almost, if ϵ is not considered).
- Transform to restore the representation power of the network: $b^{(i)} = \gamma^T x^{(i)} + \beta$
  - The parameter vectors γ and β are learnt.
- Overall: $\text{BatchNorm}_{\gamma,\beta} : a^{[k](i)} \mapsto b^{[k](i)}$
- The batch-norm transformation is differentiable, and helps in mitigating the vanishing gradient problem as well

Batch Gradient Descent

Gradient descent

Initialize weights (parameters $W^{[k]}, b^{[k]}$ for all $k = 1, \cdots, l$)

For epoch = 1, … (until stop) {

Relatively efficient with Python vectorization

Set $X = [x^{(i)}]$ and $\hat{Y} = [\hat{y}^{(i)}]$ for $i = 1, \cdots, m$ (all training examples)

Forward prop. $X \rightarrow \hat{Y} = [\hat{y}^{(i)}]$ for $i = 1, \cdots, m$

Compute cost: average of $L(y^{(i)}, \hat{y}^{(i)})$ for $i = 1, \cdots, m$

Backward prop. $\rightarrow$ compute gradients and update weights

}

- If the number of training examples is too large, each update (*descent* of the loss function) can happen only after one epoch $\Longrightarrow$ slow process

Mini-batch Gradient Descent

Gradient descent

Initialize weights (parameters $W^{[k]}, b^{[k]}$ for all $k = 1, \cdots, l$)

Mini-batch size B: $1 \leq B \leq m$

For epoch = 1, … (until stop) {

For mini-batch $b = 1, \cdots, \left\lfloor \frac{m}{B} \right\rfloor + 1$ {

Consider only the mini-batch $X_b = [x^{(i)}]$ for $i = B(b-1) + 1, \cdots, \max(Bb, m)$

Forward prop. $X_b \rightarrow \hat{Y}_b = [\hat{y}^{(i)}]$ for $i = B(b-1) + 1, \cdots, \max(Bb, m)$

Compute cost: average of $L(y^{(i)}, \hat{y}^{(i)})$ for $i = B(b-1) + 1, \cdots, \max(Bb, m)$

Backward prop. $\rightarrow$ compute gradients and update weights

}

}

examples $\Longrightarrow$ faster for reasonable choices of B

Batch, Mini-batch and Stochastic Gradient Descent

- Batch gradient descent ($B = m$)

  $\theta := \theta - \alpha \cdot \nabla_\theta J(\theta)$

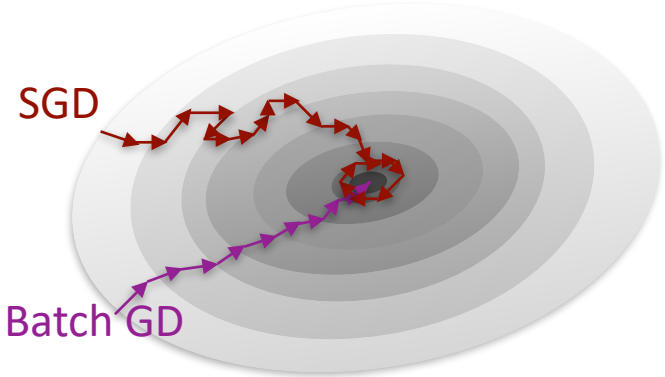
  - Guaranteed to converge
  - Computes gradients for similar examples
- Stochastic gradient descent ($B = 1$)

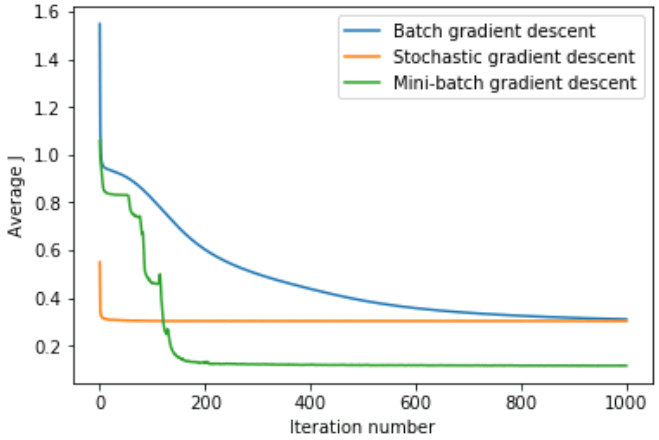
  $\theta := \theta - \alpha \cdot \nabla_\theta J(\theta; x^{(i)}; y^{(i)})$

  - Fast, can also be learned online
  - Heavy fluctuations, may never converge
- Mini-batch gradient descent (say $B = 256$)

  $\theta := \theta - \alpha \cdot \nabla_\theta J(\theta; x^{(i:i+B)}; y^{(i:i+B)})$

  - More stable than $B = 1$ (vanilla SGD)
  - Efficient, usually the choice
- Note: often, we say SGD but actually refer to mini-batch GD, because strict SGD (B = 1) is rarely used





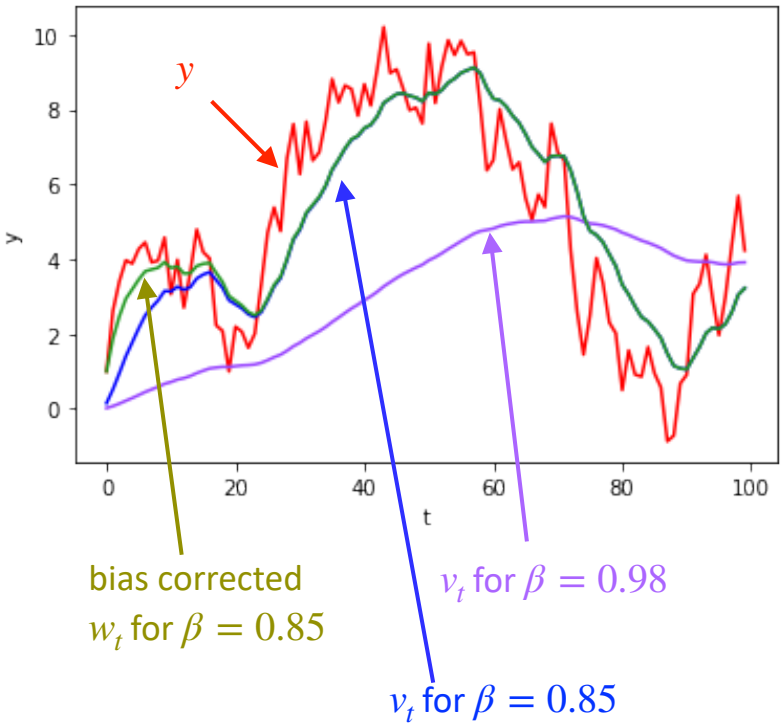
Picture source
https://adventuresinmachinelearning.com/stochastic-gradient-descent/

Challenges for Mini-batch Gradient Descent

- Choosing a proper learning rate
  - Learning rate too small: too slow to converge
  - Learning rate too large: too much fluctuations around minima, or diverge
- Reduction of learning rate, but exactly how?
  - Far from minima: it's okay to go fast
  - Near minima: switch to smaller learning rate
- Same learning rate for all weights (parameters)?
  - Sparse data, we may want to update rarely occurring features faster
- Non-convex error functions
  - Local minima
  - Saddle points (a plateau, where one dimension slopes up, another slopes down)

Moving Average (Momentum)

- Consider a sequence y_t for $t \geq 0$
- Assumption: the fluctuation contains undesired noise
- Approach: combine acquired *momentum* (till y_{t-1}) with the *present value* y_t
- Initialize: $v_{-1} = 0$
- For $t \geq 0$, $v_t = \beta v_{t-1} + (1 - \beta) y_t$
- Higher $\beta \implies$ more smoothening effect
- Bias correction: initial v_t values carry the momentum from the zero initialization
- Set $w_t = v_t / (1 - \beta^{t+1})$
- Correction particularly for small t



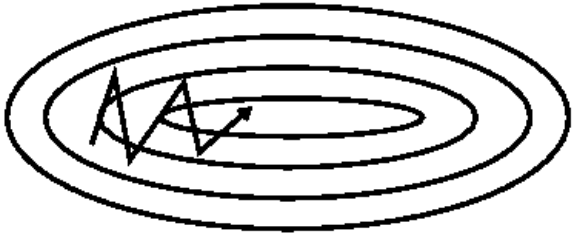
Note: the formulae change slightly if you start from $t = 1$

Gradient Descent with Momentum

- Mini-batch gradient descent: on iteration t
  - Compute $\nabla(t) = \nabla_\theta J(\theta; x^{(i:i+B)}; y^{(i:i+B)})$ on the current mini-batch (of size B)
  - Compute the moving average $v_{\nabla(t)} = \beta v_{\nabla(t-1)} + (1-\beta)\nabla(t)$
  - Update: $\theta := \theta - \alpha \cdot v_{\nabla(t)}$, instead of $\theta := \theta - \alpha \cdot \nabla$
- Bias correction is not important in this case
  - The beginning of the journey matters less
- Another intuition: dampen oscillation around the slopes of the ravine, go downhill with the momentum

without momentum

with momentum



Picture source:
https://ruder.io/optimizing-gradient-descent/

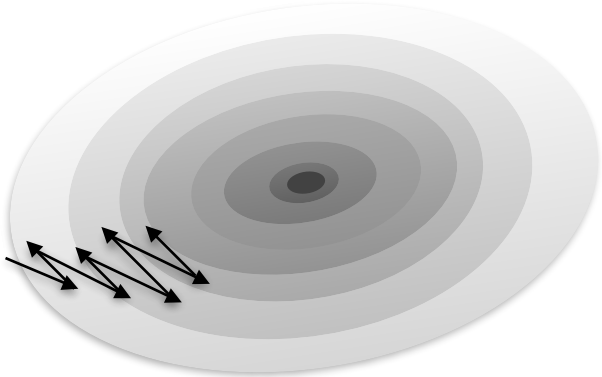
RMSProp (Root Mean Square Propagation)

- Motivation
  - The error function may have large gradient in the direction we want to move a little
  - Less steep in the direction we want to move
- On iteration t, compute the moving average of the element-wise square of the gradient

$$S_{\nabla_\theta} := \beta S_{\nabla_\theta} + (1-\beta)(\nabla_\theta * \nabla_\theta)$$

  - Update:

$$\theta := \theta - \alpha \frac{\nabla_\theta}{\sqrt{S_{\nabla_\theta}}}$$



- Intuition:
  - The update is dampened by the root-mean-square of the gradient in each direction
  - Smoothens oscillation
  - Unpublished as a research publication, proposed by G. Hinton in a Coursera course

Adaptive Moment Estimation (AdaM)

- Combining the concepts of momentum of gradient and RMSProp
- Initialize $v_{\nabla_\theta} = 0$ and $S_{\nabla_\theta} = 0$
- On each iteration t, compute using current mini-batch

$$v_{\nabla_\theta} := \beta_1 v_{\nabla_\theta} + (1 - \beta_1) \nabla_\theta$$

$$S_{\nabla_\theta} := \beta_2 S_{\nabla_\theta} + (1 - \beta_2)(\nabla_\theta * \nabla_\theta)$$

  - Apply bias correction on v_{∇_θ} and S_{∇_θ}
  - Update:

$$\theta := \theta - \alpha \frac{v_{\nabla_\theta}}{\sqrt{S_{\nabla_\theta}} + \varepsilon}$$

- Choice of parameters

$$\beta_1 = 0.9$$

$$\beta_2 = 0.999$$

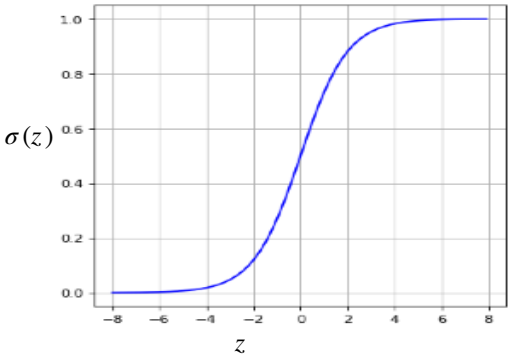
$$\varepsilon = 10^{-8}$$

α needs to be tuned

Some Activation Functions

Sigmoid

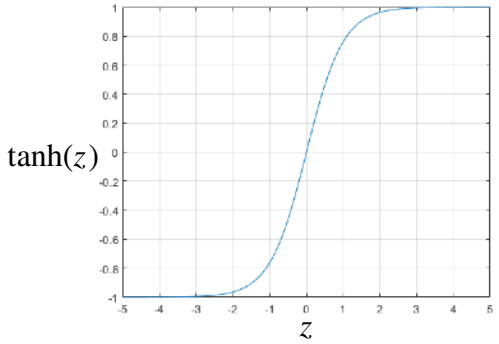
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$



- Often used for the final output layer
- Mimics binary classification (probability value)
- Cons: computing e^z is expensive

Hyperbolic tangent (tanh)

$$\tanh(z) = \frac{1 - e^{2z}}{1 + e^{2z}}$$

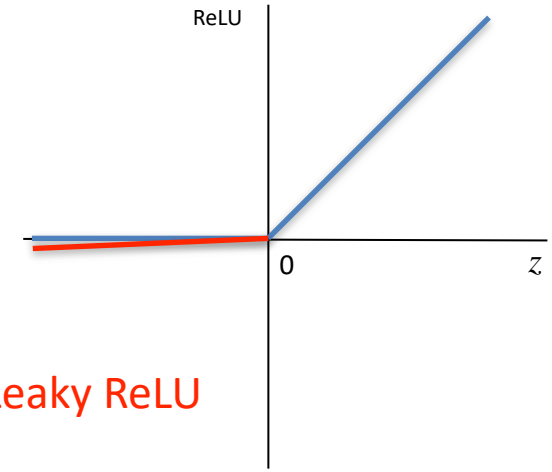


- Bound to the range $(-1,1)$
- Gradient steeper than sigmoid, hence optimizes faster
- Cons: Similar to sigmoid, has vanishing gradient problem

Some Activation Functions

ReLU

$$g(z) = \max\{0, z\}$$



Leaky ReLU

$g(z) = \epsilon z$ if $z < 0$

for some small constant ϵ

say $\epsilon = 0.001$

- Fast to compute (no expensive operation)
- Faster convergence (up to 6 times faster than sigmoid or tanh)
- Solves the vanishing gradient problem
  - Has slope 1 for values > 0, so does not cause the gradient to vanish
- Output values can be large, not used much in RNN (values can explode)
- Dying ReLU: Once a neuron goes to negative, then set to 0, it may never recover
- Leaky ReLU: solving the dying ReLU by allowing a small slope towards negative for $z < 0$

Deep feedforward networks (l - layers)

Input layer 0

$a^{[0]} = x$

Layer 1

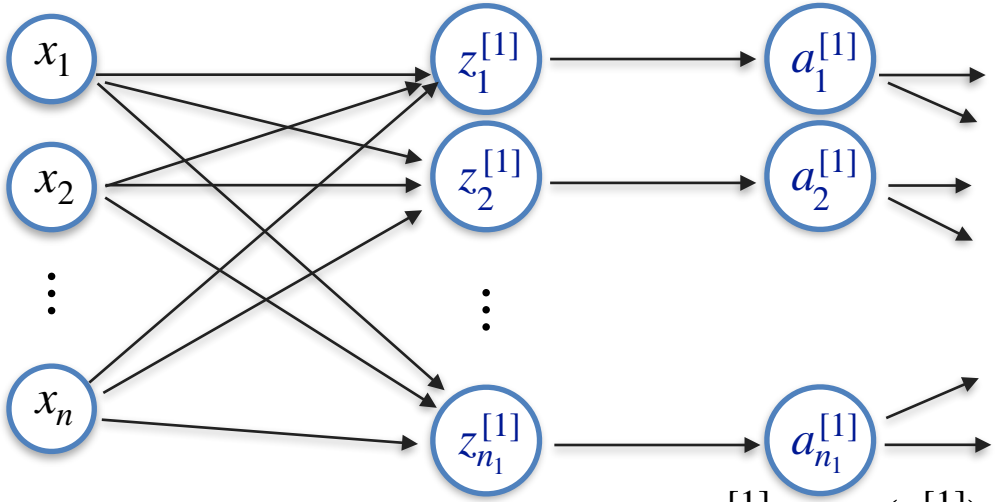
$z^{[1]}$

Layer 1

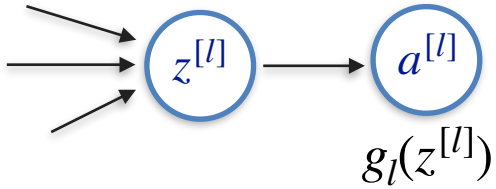
$a^{[1]}$

Output layer l

$a^{[l]} = \hat{y}$



Hidden layers



$z^{[1]} = W^{[1]}a^{[0]} + b^{[1]}$

affine transformation

$a_i^{[1]} = g_1(z_i^{[1]})$

non-linear activation g_1 (element-wise)

$z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]}$

Each layer: an affine transformation, followed by a non-linear activation

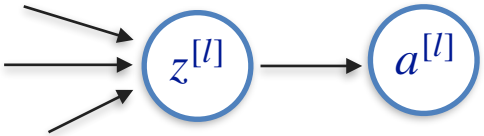
Softmax Activation

- For binary classification, we let the output layer to have a single neuron (1×1)
- For multiclass classification, we can have the output layer to have dimension $c \times 1$, where c is the number of classes
- Softmax activation:

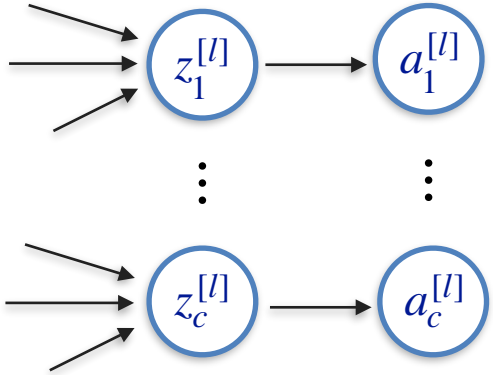
$$g(z) = \frac{e^{z_i}}{\sum_{i=1}^{c} e^{z_i}}$$

- Intuition:
  - Generalization of sigmoid
  - Each activation is a probability
  - Differentiable on real numbers

Output layer with a single neuron



Output layer with c neurons



Transfer learning

- A basic assumption is supervised learning: the training samples and test samples are drawn from the same *distribution*
- Intuitively: same task on same domain
- Random initialization of a model: training the model from the scratch
  - Is that necessary?
- Human analogy (not all would perfectly resonate)
  - Learn to drive in the city $\Longrightarrow$ can adapt (with little training) in the hills
  - Learn mathematics $\Longrightarrow$ learn computer science relatively easily
  - Learn English $\Longrightarrow$ learn German (compared to those who don't know English) easily
- Many approaches in transfer learning (very important subfield of research)
  - Freeze initial layers and only fine - tune final layers
  - Gradually unfreeze layers, slow down learning rate

References

- Andrew Ng's lectures on *Improving Deep Neural Networks: Hyperparameter tuning, Regularization and Optimization*: www.coursera.org/learn/deep-neural-network
- Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016. www.deeplearningbook.org
- Ruder, Sebastian. *An overview of gradient descent optimization algorithms*. *arXiv preprint arXiv:1609.04747* (2016). arxiv.org/abs/1609.04747
- Pal, Sayak: *Towards preventing overfitting: regularization*, 2018. www.datacamp.com/community/tutorials/towards-preventing-overfitting-regularization
- Tieleman, Tijmen, and Geoffrey Hinton. *Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude*. *COURSERA: Neural networks for machine learning* 4, no. 2 (2012): 26-31. https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf