Day 7: Optimization, regularization for NN

Introduction to Machine Learning Summer School
June 18, 2018 - June 29, 2018, Chicago

Instructor: Suriya Gunasekar, TTI Chicago

26 June 2018

TOYOTA TECHNOLOGICAL INSTITUTE AT CHICAGO

THE UNIVERSITY OF CHICAGO

NSF

Day 7: Tricks and tools for NN training

Introduction to Machine Learning Summer School
June 18, 2018 - June 29, 2018, Chicago

Instructor: Suriya Gunasekar, TTI Chicago

26 June 2018

TOYOTA TECHNOLOGICAL INSTITUTE AT CHICAGO

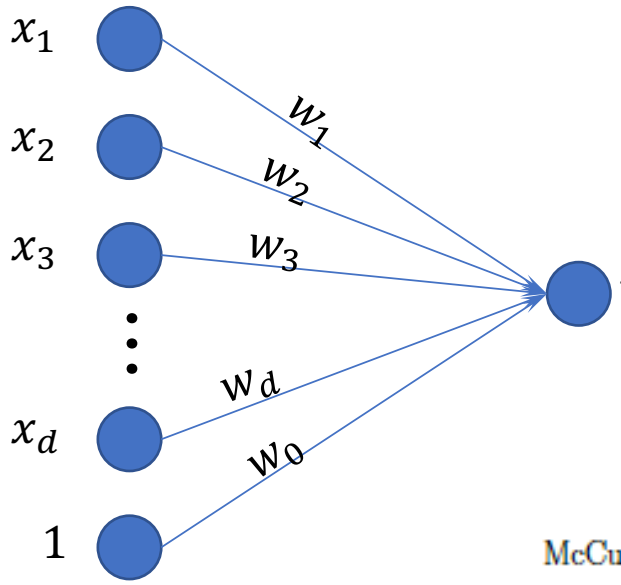
THE UNIVERSITY OF CHICAGO

NSF

Topics so far

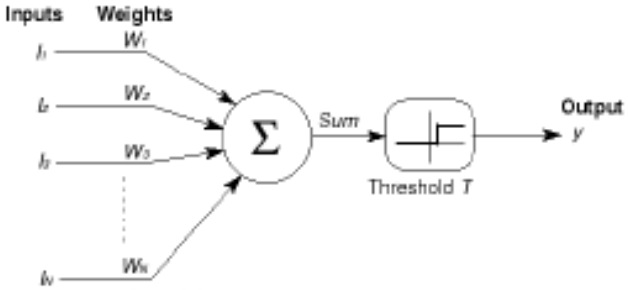
- Linear regression
- Classification
  - Logistic regression
  - Maximum margin classifiers, kernel trick
  - Generative models
- Yesterday
  - Neural networks introduction
  - Backpropagation
- Today
  - Common practices in NN training – optimization and regularization
  - Special architectures – CNNs, RNNs, encoder-decoder

Linear classifier



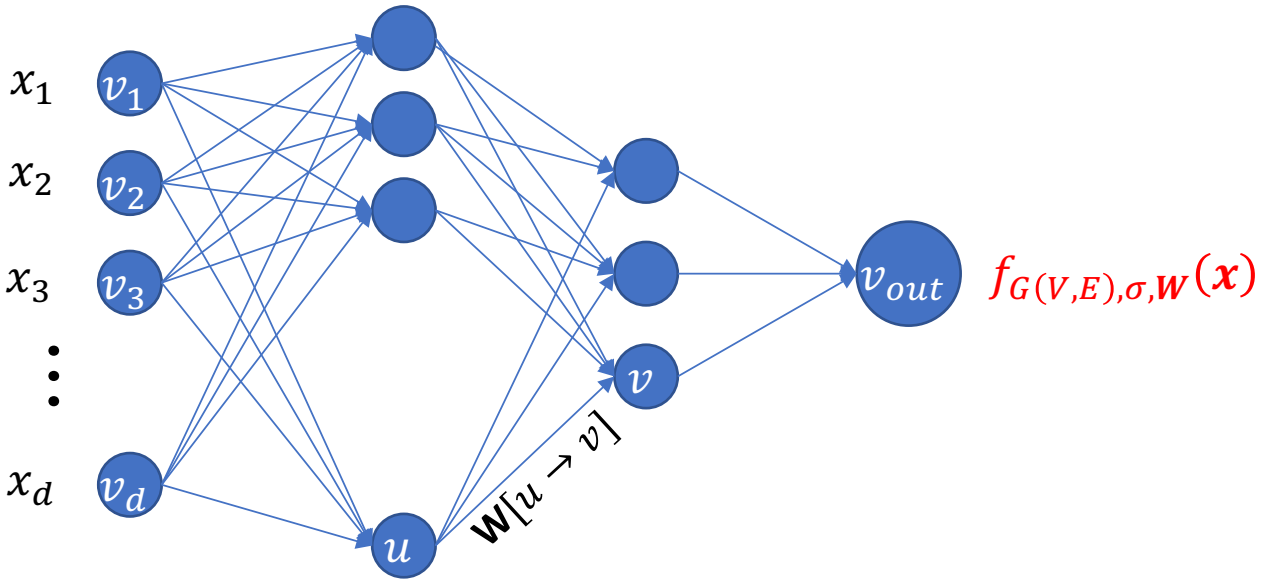
$$f(\boldsymbol{x}) = \mathbf{1}(\boldsymbol{w}.\boldsymbol{x} + w_0 \geq 0)$$

McCullock and Pitts 1943 — introduced the linear threshold "neuron".



- Biological analogy: single neuron – stimuli reinforce synaptic connections

Feed-Forward Neural Networks

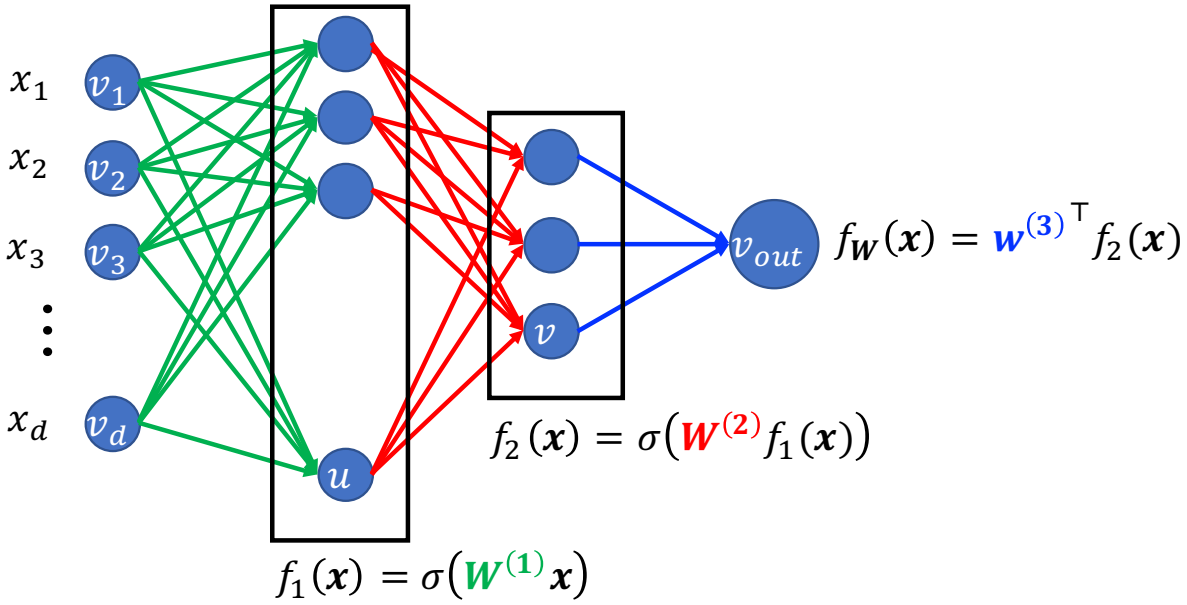


Architecture:

- Directed Acyclic Graph G(V,E). Units (neurons) indexed by vertices in V.
  - "Input Units" $v_1 \ldots v_d \in V$: no incoming edges have value $o[v_i] = x_i$
  - Each edge $u \to v$ has weight $\boldsymbol{W}[u \to v]$
    - Pre-activation $a[v] = \sum_{u \to v \in E} \boldsymbol{W}[u \to v]\, o[u]$
    - Output value $o[v] = \sigma(\, a[v]\,)$
  - "Output Unit" $v_{out} \in V$, $f_W(\boldsymbol{x}) = a[v_{out}]$

Figure credit: Nati Srebro

Feed forward fully connected network



- L hidden layers with layer l havinb d_l hidden units
- Parameters:
  - for each intermediate layer $\boldsymbol{W}^{(l)} \in \mathbb{R}^{d_{l-1} \times d_l}$ where $d_0 = d$
  - final layer weights $\boldsymbol{w}^{(L+1)} \in \mathbb{R}^{d_L}$
- For 2-hidden layer $f_W(\boldsymbol{x}) = \boldsymbol{w}^{(3)^\top} \sigma\left(\boldsymbol{W}^{(2)} \sigma\left(\boldsymbol{W}^{(1)}\boldsymbol{x}\right)\right)$. More generally,

$$f_W(\boldsymbol{x}) = \boldsymbol{w}^{(L+1)^\top} \sigma\left(\boldsymbol{W}^{(L-1)} \ldots \sigma\left(\boldsymbol{W}^{(2)} \sigma\left(\boldsymbol{W}^{(1)}\boldsymbol{x}\right)\right)\right)$$

Neural networks as hypothesis class

- Hypothesis class specified by:
  - Graph G(V,E)
  - Activation function σ

  } Based on architecture and fixed

  - Weights $\mathbf{W}$, with weight $\mathbf{W}[u \to v]$ for each edge $u \to v \in E$

$$\mathcal{H} = \left\{ f_{G(V,E),\sigma,\mathbf{W}} \mid \boldsymbol{W}: E \to \mathbb{R} \right\}$$

- Expressive power:

$$\{ f \mid f \text{ computable in time } T \} \subseteq \mathcal{H}_{G(V,E),sign} \quad \text{with } |E| = O(T^2)$$

  - demo
- Computation: empirical risk minimization

$$\widehat{\mathbf{W}} = \arg\min_{W} \sum_{i=1}^{N} \ell\left(f_{G(V,E),\sigma,\boldsymbol{W}}\left(\boldsymbol{x}^{(i)}\right), y^{(i)}\right)$$

  - Highly non-convex problem, even if $loss\ \ell$ is convex
  - Hard to minimize over even tiny neural networks are hard
    - If not supervised ML will be solved

SGD

$$\widehat{\boldsymbol{W}} = \arg\min_{W} \sum_{i=1}^{N} \ell\left(f_{\boldsymbol{W}}\left(\boldsymbol{x}^{(i)}\right), y^{(i)}\right)$$

- Stochastic gradient descent: for random $\left(\boldsymbol{x}^{(i)}, y^{(i)}\right) \in S$

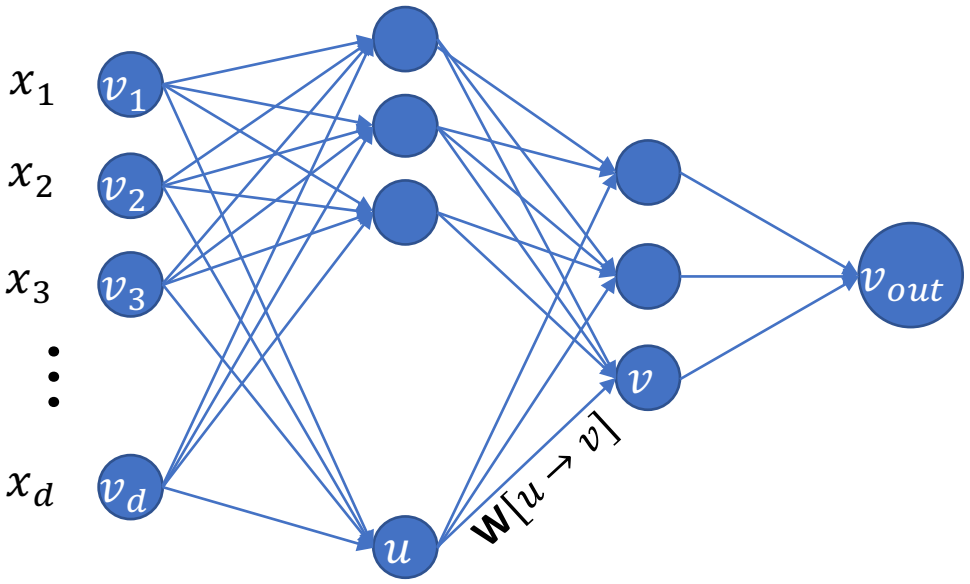
$$\boldsymbol{W}^{(t+1)} \leftarrow \boldsymbol{W}^{(t)} - \eta^{(t)} \nabla \ell\left(f_{\boldsymbol{W}^{(t)}}\left(\boldsymbol{x}^{(i)}\right), y^{(i)}\right)$$

  (Even though its not convex)

- How to calculate $\nabla \ell\left(f_{\boldsymbol{W}^{(t)}}\left(\boldsymbol{x}^{(i)}\right), y^{(i)}\right)$?
  Backpropagation → chain rule + dynamic programming
  - Computing gradients as easy (or hard) as computing the function itself
  - What about memory?
  - Once you define gradients over simple operations, can easily compose to get complex gradients
    - Idea behind autograd

Back-Propagation

- Efficient calculation of $\nabla_{\boldsymbol{W}} \ell(f_{\boldsymbol{W}}(\boldsymbol{x}), y)$ using chain rule



$$a[v] = \sum_{u \to v \in E} \boldsymbol{W}^{(t)}[u \to v]\, o[u]$$

$$o[v] = \sigma(\, a[v]\,)$$

$$z[v_{out}] = \ell'(a[v_{out}], y)$$

$$z[u] = \sigma'(a[u]) \sum_{u \to v} \boldsymbol{W}^{(t)}[u \to v] z[v]$$

- Forward propagation: calculate activations $a[v]$ and outputs $o[v]$
- Backward propagation: calculate $z[v] \stackrel{\text{def}}{=} \frac{\partial \ell(f_{\boldsymbol{W}}(\boldsymbol{x}), y)}{\partial \boldsymbol{a}[v]}$
- Gradient descent update: using $\frac{\partial \ell(f_{\boldsymbol{W}}(\boldsymbol{x}), y)}{\partial W^{(t)}[u \to v]} = z[v] o[u]$

$$\boldsymbol{W}^{(t+1)}[u \to v] = \boldsymbol{W}^{(t+1)}[u \to v] - \eta^{(t)} \frac{\partial \ell(f_{\boldsymbol{W}}(\boldsymbol{x}), y)}{\partial W^{(t)}[u \to v]}$$

Putting it All Together: SGD on Neural Networks

- Initialize $\boldsymbol{W}^{(0)}$ randomly (small, but not zero)
- For $t = 1,2, \ldots$:
  - Sample $\left(\boldsymbol{x}^{(i)}, y^{(i)}\right)$ (from tanning set S)
  - Calculate the gradient $\boldsymbol{g}^{(t)}$ using backpropagation
    - Forward pass: traverse the graph forward (starting from input units) and calculate activation and output values

$$a[v] = \sum_{u \to v \in E} \boldsymbol{W}^{(t)}[u \to v]\, o[u] \quad \text{and} \quad o[v] = \sigma(\, a[v]\,)$$

    - Backward pass: Calculate gradients with respect to activations by scanning the graph backward starting with output note

$$z[v_{out}] = \frac{\partial \ell\left(a[v_{out}], y^{(i)}\right)}{\partial a[v_{out}]} \text{ and } z[u] = \sigma'(a[u]) \sum_{u \to v \in E} \boldsymbol{W}^{(t)}[u \to v] z[v]$$

    - Output gradients with respect to weights

$$\boldsymbol{g}^{(t)}[u \to v] = \frac{\partial \ell\left(f_{\boldsymbol{W}^{(t)}}\left(\boldsymbol{x}^{(i)}\right), y^{(i)}\right)}{\partial \boldsymbol{W}[u \to v]} = z[v] o[u]$$

  - Update weights: $\boldsymbol{W}^{(t+1)} = \boldsymbol{W}^{(t)} - \eta^{(t)} \boldsymbol{g}^{(t)}$

History of Neural Networks

- 1940s-60s:
  - Inspired by learning in the brain, and as a model for the brain (Pitts, Hebb, and others)
  - Various models, directed and undirected, different activation and learning rules
- 1960-70s:
  - Perceptron Rule (Rosenblatt), Multilayer perceptron (Minksy and Papert): that many properties of images could not be determined by (single layer) perceptron. Caused a decline of activity in neural networks.
- 1970s-early 1980s:
  - Backpropagation (Werbos 1975), practical backprop (Rumelhart, Hinton et al 1986) and SGD (Bottou)
  - Initial empirical success
- 1980s-2000s:
  - Lost favor to implicit linear methods like SVM and boosting with convex losses and convex relaxations
  - Also time when much of tools in todays deep learning is discovered –CNNs, LSTMs, etc.
- 2000-2010s:
  - revival of interest (CIFAR groups)
  - layer-wise pre-training of deep-ish nets were being trainined
  - progress in speech and vision with deep neural nets
- 2010s:
  - Computational advances (and also a few new tricks) allow training large networks
  - 2012: Krizhevsky et al. win ImageNet
  - Empirical success and renewed interest

History of Neural Networks

- 1940s-60s:
  - Inspired by learning in the brain, and as a model for the brain (Pitts, Hebb, and others)
  - Various models, directed and undirected, different activation and learning rules
- 1960-70s:
  - Perceptron Rule (Rosenblatt), Multilayer perceptron (Minksy and Papert): that many properties of images could not be determined by (single layer) perceptron. Caused a decline of activity in neural networks.
- 1970s-early 1980s:
  - Backpropagation (Werbos 1975), practical backprop (Rumelhart, Hinton et al 1986) and SGD (Bottou)
  - Initial empirical success
- 1980s-2000s:
  - Lost favor to implicit linear methods like SVM and boosting with convex losses and convex relaxations
  - Also time when much of tools in todays deep learning is discovered –CNNs, LSTMs, etc.
- 2000-2010s:
  - revival of interest (CIFAR groups)
  - layer-wise pre-training of deep-ish nets were being trainined
  - progress in speech and vision with deep neural nets
- 2010s:
  - Computational advances (and also a few new tricks) allow training large networks
  - 2012: Krizhevsky et al. win ImageNet
  - Empirical success and renewed interest

So what changed?

- Large datasets
  - Computer vision: classification datasets →CIFAR ~60K images, ImageNet ~14M images, ~1M annotations!
    - similarly large datasets for other vision tasks like segmentation, detection etc.
  - Game playing: Go, Chess → can simulate as much data as allowed by compute power
  - NLP and speech (ask Karl, I don't know!)
  - Other domains again data collection and storage is much cheaper
- Advances in computation
  - Advancement in GPU technology,
  - SGD training on GPUs for #weights ≈ #samples ≈ $10^7 \sim 10^9$ or more
  - Optimization technology: momentum, AdaGrad, normalization
  - What's constant since the 50s: training runtime is about 10-14 days!
- Other tools/tricks
  - Non-saturating activation: $\sigma(z) = [z]_+ = ReLU(z) = \max(0, z)$
  - Newish regularization techniques like dropout
  - Pre-training leading to efficient transfer learning

Optimization

Neural network training

$$\widehat{\boldsymbol{W}} = \arg\min_{W} \sum_{i=1}^{N} \ell\left(f_{\boldsymbol{W}}\left(\boldsymbol{x}^{(i)}\right), y^{(i)}\right)$$

- Stochastic gradient descent: for random $\left(\boldsymbol{x}^{(i)}, y^{(i)}\right) \in S$

$$\boldsymbol{W}^{(t+1)} \leftarrow \boldsymbol{W}^{(t)} - \eta^{(t)} \nabla \ell\left(f_{\boldsymbol{W}^{(t)}}\left(\boldsymbol{x}^{(i)}\right), y^{(i)}\right)$$

  (Even though its not convex)

- Use backprop to calculate $\nabla \ell\left(f_{\boldsymbol{W}^{(t)}}\left(\boldsymbol{x}^{(i)}\right), y^{(i)}\right)$!

- What could go wrong?
- What are the other options?

SGD common pitfalls

- For some random sample $(\boldsymbol{x}, y)$, you write a program to compute $\nabla_{\boldsymbol{W}} \ell(f_{\boldsymbol{W}}(\boldsymbol{x}), y)$!
- What is the first mistake that can happen?

Gradient computation

- For some random sample $(\boldsymbol{x}, y)$, you write a program to compute $\nabla_{\boldsymbol{W}} \ell(f_{\boldsymbol{W}}(\boldsymbol{x}), y)$!
- What is the first mistake that can happen?
  - Wrong gradient program!
  - How to fix/avoid?
  - Write a numerical gradient checker (hopefully without bugs).

    Recall $\nabla_w f(w)[i] = \frac{\partial f(w)}{\partial w_i} = \lim_{\delta \to 0} \frac{f(w+\delta e_i) - f(w)}{\delta} = \lim_{\delta \to 0} \frac{f(w+\delta e_i) - f(w-\delta e_i)}{2\delta}$
  - $\text{GradCheck}(f, \text{grad}_f, w)$:
    - $g_{\text{grad}_f} = \text{grad}_f(w)$
    - for $i = 1,2, \ldots$
      - $g_{numeric}[i] = \frac{f(w+\delta e_i) - f(w-\delta e_i)}{2\delta}$
      - if $la.norm\left(g_{\text{grad}_f}[i] - g_{numeric}[\text{i}]\right) > \epsilon$: **error**

Good idea to add whenever you program gradient computation manually

SGD common pitfalls

- For some random sample $(\boldsymbol{x}, y)$, you write a program to compute $\nabla_{\boldsymbol{W}} \ell(f_{\boldsymbol{W}}(\boldsymbol{x}), y)$!
- What is the first mistake that can happen?
  - Wrong gradient program! → Write a numerical gradient checker
- Less obvious mistake! What is the order of samples we get in SGD?
  - Does it matter? Can't we just cycle through the data?
    - What happens if all the cats are stored first and then all the dogs?
  - Ideally: Use fresh sample at each iteration $(x, y) \sim \mathcal{D}$
    - In practice, we have to reuse samples from training set

SGD common pitfalls

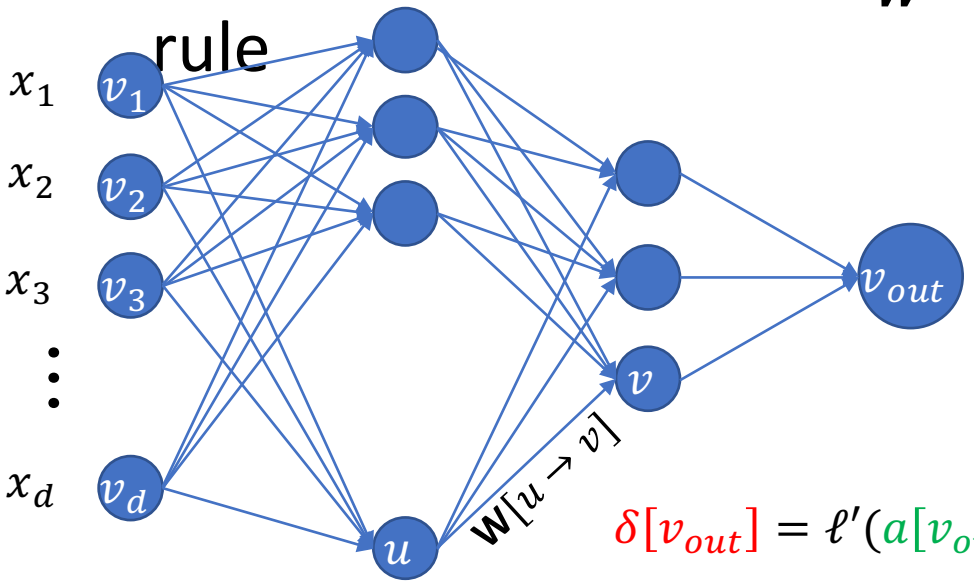
- For some random sample $(\boldsymbol{x}, y)$, you write a program to compute $\nabla_{\boldsymbol{W}} \ell(f_{\boldsymbol{W}}(\boldsymbol{x}), y)$!
- What is the first mistake that can happen?
  - Wrong gradient program! → Write a numerical gradient checker
- Less obvious mistake! What is the order of samples we get in SGD?
  - Does it matter? Can't we just cycle through the data?
    - What happens if all the cats are stored first and then all the dogs?
  - Ideally: Use fresh sample at each iteration $(x, y) \sim \mathcal{D}$
    - In practice, we have to reuse samples from training set
  - Ok! Can we just sample independently at each iteration (with replacement)?
    - Better: sample without replacements, but remember to randomly permute then cycle
    - Best: for each pass over data ("epoch"), use different **random** order

Optimization

- Common problems arising from models
  - Gradient clipping
  - Gradient explosion
- SGD “knobs” in NN training
  - Initialization
  - Step-size
  - SGD variants
    - Momentum for SGD
    - Adaptive variants of SGD
  - Mini-batch SGD
  - Batch normalization

Back-Propagation

- Efficient calculation of $\nabla_{\boldsymbol{W}} \ell(f_{\boldsymbol{W}}(\boldsymbol{x}), y)$ using chain rule



$$a[v] = \sum_{u \to v \in E} \boldsymbol{W}^{(t)}[u \to v]\, o[u]$$

$$o[v] = \sigma(\, a[v]\,)$$

$$\delta[v_{out}] = \ell'(a[v_{out}], y)$$

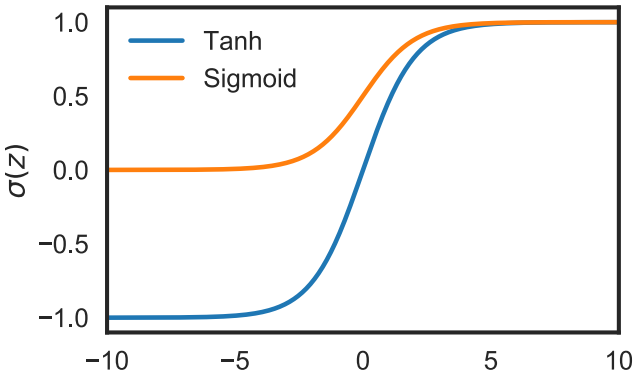
$$\delta[u] = \sigma'(a[u]) \sum_{u \to v} \boldsymbol{W}^{(t)}[u \to v]\, \delta[v]$$

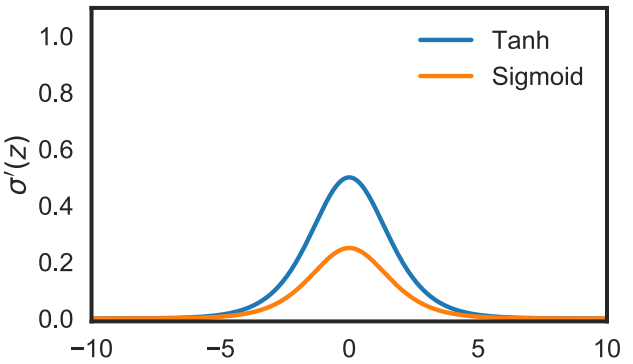
$$\sigma'(z) = \frac{\mathrm{d}\sigma(z)}{\mathrm{d}z}$$

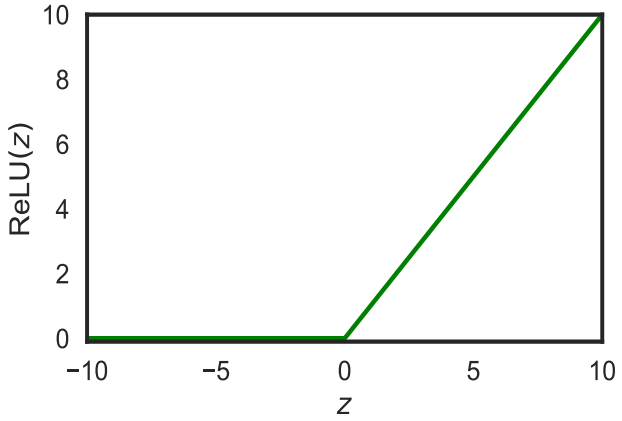
Figure credit: Nati Srebro

Activation functions

- Sigmoid $\sigma(z) = \frac{1}{1+\exp(-z)}$
- Tanh $\sigma(z) = \frac{1-\exp(-z)}{1+\exp(-z)}$
  - The good:
    - squash outputs to a fixed range
    - no gradient explosion from repeatedly multiplying $\boldsymbol{W}^{(l)}$
  - The bad
    - gradient $\sigma'(z)$ is nearly zero for most values of z
- ReLU $\sigma(z) = \max(0, z)$
  - Avoids gradient saturation (in part), but can lead to gradient explosion in some architectures (e.g., RNNs)
    - Gradient clipping - $g^{(t)} = \max(g^{(t)}, G_{\max})$







Activation functions

- If during SGD updates, a ReLU unit gets to a state where for all data points, the activations is 0, then the unit never recovers from 0 gradient
- Some variants of ReLU
  - Leaky ReLU: $\sigma(z) = \max(\alpha z, z)$ where $\alpha > 1$
  - Exponential ReLU $\sigma(z) = \begin{cases} z & \text{if } z \geq 0 \\ \alpha(\exp(z) - 1) & \text{if } z < 0 \end{cases}$

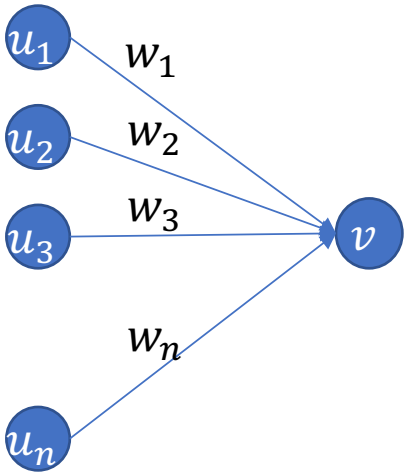
Optimization

- Common problems arising from models
  - Gradient clipping
  - Gradient explosion
- SGD “knobs” in NN training
  - Initialization
  - Step size/learning rate
  - SGD variants
    - Momentum for SGD
    - Adaptive variants of SGD
  - Mini-batch SGD
  - Batch normalization

Knob 1: Initialization

Non-convex objective: initialization plays a crucial role

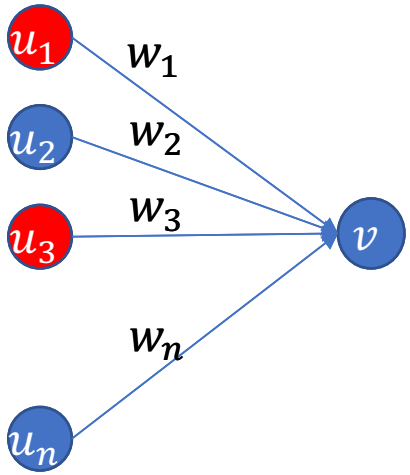
- Can we initialize all weights to 0?
- Random initialization: Initialize all weights with small random real numbers, e.g., Gaussian with mean zero, $\mathcal{N}(0,0.01)$
  - Consider a node v with n incoming weights w_i
  - Assume parent nodes are also mean zero.
    What is variance of activation $a[v]$ at node v
    with incoming weights $w_i \sim \mathcal{N}(0,0.01)$?



$$var(a[v]) = var(\sum_i w_i\, u_i) =$$
$$\#parents.\, var(u_i) var(w_i)$$

Knob 1: Initialization

- Xavier initialization: scale the std-dev to normalize the variance in each node
  - if **node v has n incoming weights**, each incoming weight gets random initialization of $\mathcal{N}(0, \sigma^2/n)$
  - This assumes parent nodes are zero mean
  - what values can parent nodes take after activation?



$$o[u] = \sigma(a[u])$$

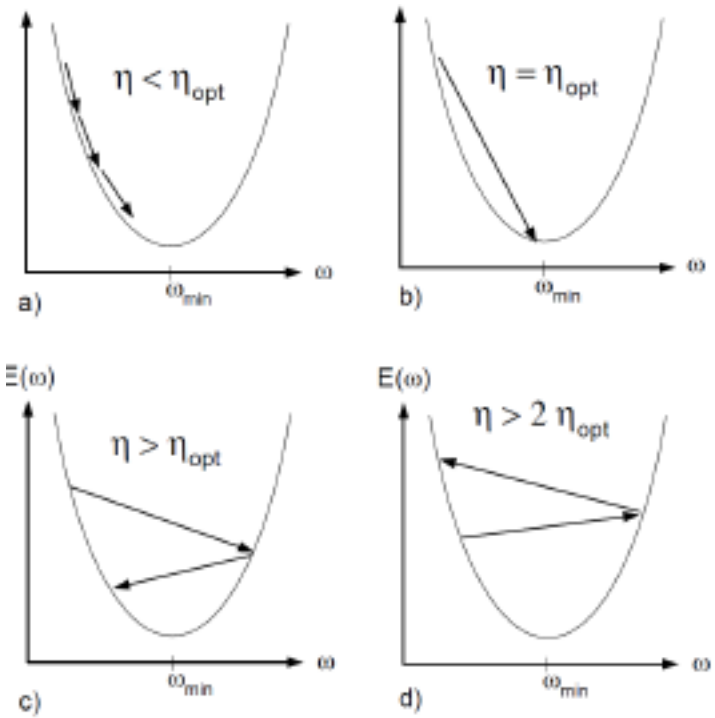
    - was proposed for zero mean activations: not satisfied by ReLUs

- Kaiming initialization: specifically for ReLUs.
  - On avg. we will have half of the units active, so initialize incoming weights of node v with $\mathcal{N}(0, 2\sigma^2/n)$

Knob 2: Step size/learning rate

Learning rate/step η_t size is the most important parameter to tune

- Theory from convex optimization: for SGD decay the learning rate with t as $\approx \frac{1}{c+t}$ → Use only as heuristic – does not extent for non-convex function



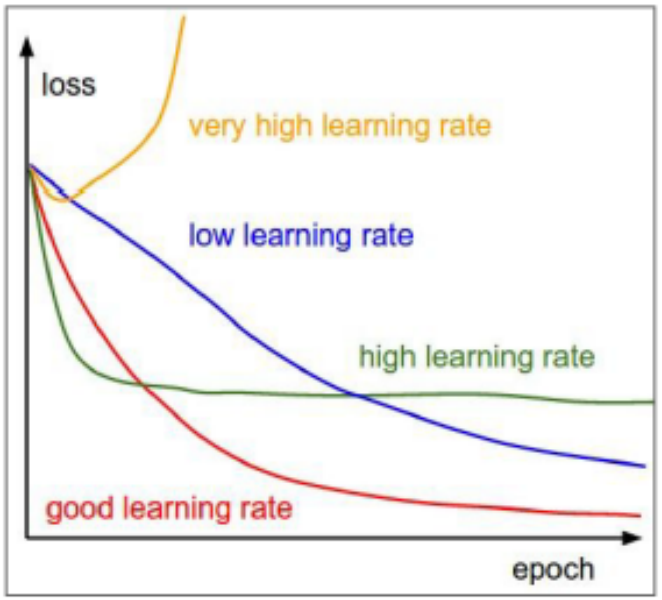


Figure credit: Lecun et al. (1996), Andrej Karpathy, Greg Shaknarovich

Knob 2: Step size/learning rate

Learning rate/step size η_t is the most important parameter to tune

- In practice: some degree of babysitting
  - start with a reasonable step size, $\eta_t = 0.01$
  - monitor validation/training loss
  - drop η_t (typically 1/10) when learning appears stuck
- Tips
  - wait a bit before dropping;
  - If monitoring training loss,
    - calculating loss on full dataset can be expensive
    - instead use moving average from SGD iterations
  - Crashes due to NaNs etc. often due to η_t

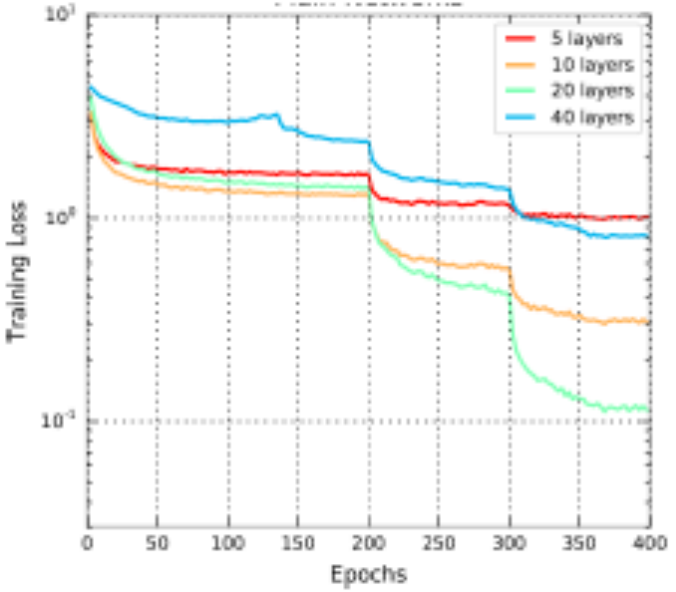


Figure credit: Larson et al.

Knob 3: Variants of SGD

$$\widehat{\boldsymbol{W}} = \arg\min_{W} \sum_{i=1}^{N} \ell\left(f_{\boldsymbol{W}}\left(\boldsymbol{x}^{(i)}\right), y^{(i)}\right)$$

- Stochastic gradient descent: for random $\left(\boldsymbol{x}^{(i)}, y^{(i)}\right) \in S$

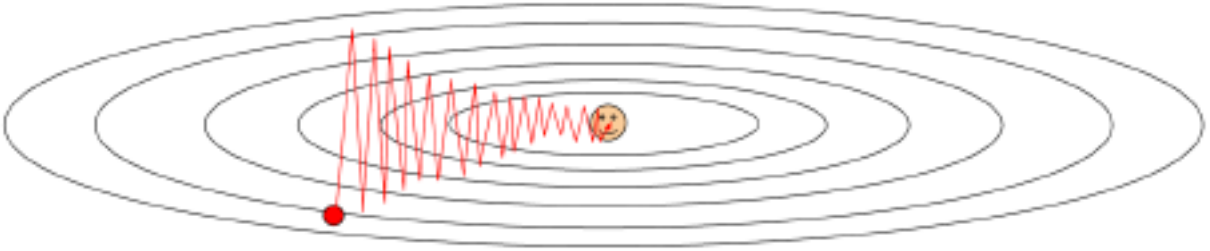
$$\boldsymbol{W}^{(t+1)} \leftarrow \boldsymbol{W}^{(t)} - \eta^{(t)} \nabla \ell\left(f_{\boldsymbol{W}^{(t)}}\left(\boldsymbol{x}^{(i)}\right), y^{(i)}\right)$$

- optim.SGD(model.parameters(), lr = 0.01)

- Two variants of SGD are commonly used:
  - Momentum
    - optim.SGD(model.parameters(), lr = 0.01, momentum=0.9)
  - Adaptive step sizes
    - Adagrad: optim.Adagrad(model.parameters(), lr = 0.01)
    - Adam: optim.Adam(params, lr=0.001, betas=(0.9, 0.999))

Knob 3a: Momentum for SGD

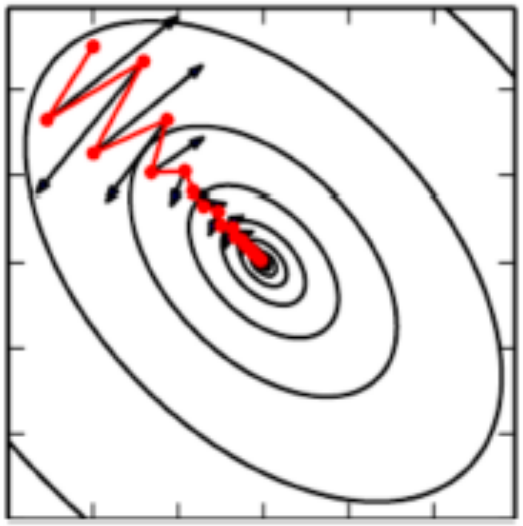
- S(GD) have trouble navigating areas where the curvature is steeper in one dimension than the other
  - ends up oscillating around the slopes and makes slow progress



- Fix: Momentum term

$$W^{(t+1)} = W^{(t)} - \eta^{(t)} \nabla \ell(f_W(x), y) + \gamma^{(t)}\left(W^{(t)} - W^{(t-1)}\right)$$

  - reduces updates along directions that change gradients frequently
  - increases updates along directions where the gradients are consistent
  - dampens oscillations



[Goodfellow et al]

Figure credit: Andrej Karpathy, Greg Shaknarovich

Knob 3b: Adaptive step sizes

$$\boldsymbol{W}^{(t+1)}[u \to v] = \boldsymbol{W}^{(t)}[u \to v] - \eta_t \boldsymbol{g}^{(t)}[u \to v]$$

- All weights have same learning rate

AdaGrad: Reduce learning rate proportional to updates.

$$\boldsymbol{s}^{(t)}[u \to v] = \boldsymbol{s}^{(t-1)}[u \to v] + \left(\boldsymbol{g}^{(t)}[u \to v]\right)^2$$

$$\boldsymbol{W}^{(t+1)}[u \to v] = \boldsymbol{W}^{(t)} - \frac{\eta_t}{\sqrt{\boldsymbol{s}^{(t)}[u \to v] + \epsilon}} \boldsymbol{g}^{(t)}[u \to v]$$

- Rarely used, reduces learning rate too aggressively

RMSprop: Adagrad + forgetting

$$\boldsymbol{s}^{(t)}[u \to v] = \delta \boldsymbol{s}^{(t-1)}[u \to v] + (1-\delta)\left(\boldsymbol{g}^{(t)}[u \to v]\right)^2$$

$$\boldsymbol{W}^{(t+1)}[u \to v] = \boldsymbol{W}^{(t)} - \frac{\eta_t}{\sqrt{\boldsymbol{s}^{(t)}[u \to v] + \epsilon}} \boldsymbol{g}^{(t)}[u \to v]$$

Knob 3b: Adaptive step sizes

$$\boldsymbol{W}^{(t+1)}[u \to v] = \boldsymbol{W}^{(t)} - \eta_t \boldsymbol{g}^{(t)}[u \to v]$$

Adam: RMSprop with momentum

$$\boldsymbol{m}^{(t)}[u \to v] = \beta_1 \boldsymbol{m}^{(t-1)}[u \to v] + (1-\beta_1)\boldsymbol{g}^{(t)}[u \to v]$$

$$\boldsymbol{s}^{(t)}[u \to v] = \beta_2 \boldsymbol{s}^{(t-1)}[u \to v] + (1-\beta_2)\left(\boldsymbol{g}^{(t)}[u \to v]\right)^2$$

$$\boldsymbol{W}^{(t+1)}[u \to v] = \boldsymbol{W}^{(t)} - \frac{\eta_t}{\sqrt{\boldsymbol{s}^{(t)}[u \to v]} + \epsilon}\boldsymbol{m}^{(t)}[u \to v]$$

- Most commonly used adaptive method.
  - optim.Adam(params, lr=0.001, betas=(0.9, 0.999))
- Good first step:
  - Pick one of (SGD+momentum) or (Adam)

Knob 4: Mini-batches

- Instead of using a single example to obtain gradient estimate, use multiple examples:
- Pick m examples $B^{(t)} = \left\{ i_1^{(t)}, i_2^{(t)}, \dots, i_m^{(t)} \right\}$ randomly

$$g^{(t)} = \frac{1}{m} \sum_{i \in B^{(t)}} \nabla_w \ell\left(f_{\boldsymbol{W}^{(t)}}\left(\boldsymbol{x}^{(i)} \right), y^{(i)} \right)$$

☺ At each iteration: better gradient estimate, better (more accurate) update step

☹ But at the cost of m backprops per update

Allows parallelization, pipelining, efficient memory access

Knob 5: (Mini)Batch Normalization

$$o[v] = \sigma\left(c_v \frac{\left(a[v] - \widehat{\mathbb{E}}[a[v]]\right)}{\sqrt{\widehat{\mathrm{Var}}[a[v]]}} + b_v\right)$$

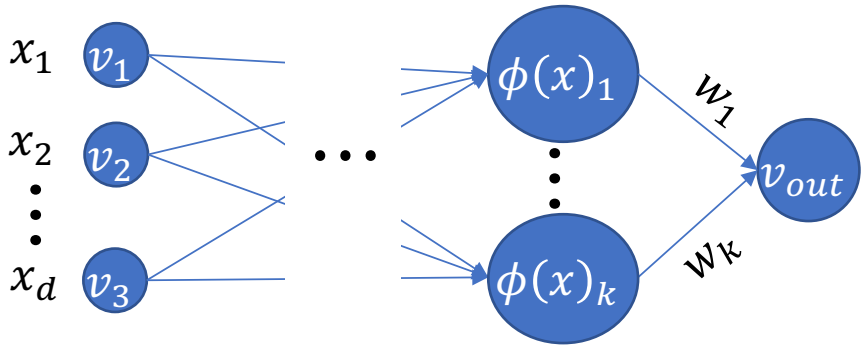
Calculated on minibatch

- Different parametrization of same function class
- SGD (or AdaGrad or ADAM) on $\{\boldsymbol{W}, \{c_v\}, \{b_v\}\}$
- Greatly helps with optimization in practice

Slide credit: Nati Srebro

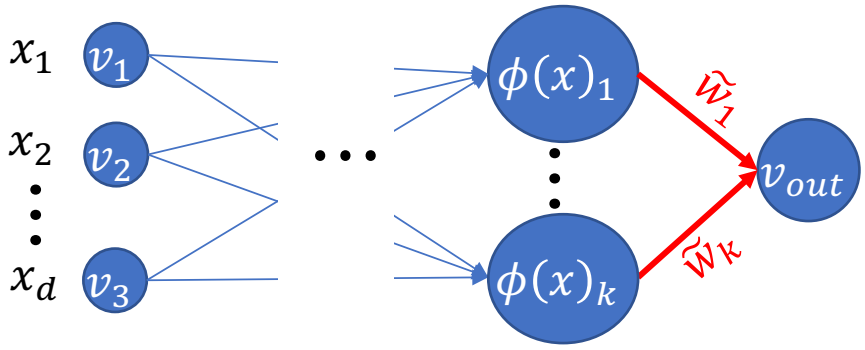
Bonus knob: warm start/pre-training

- Suppose we want to continue training for more epochs
  - save snapshots of weights and resume again
  - need to carefully initialize learning rate now
- Also, can use weights pre-trained from another task as initialization for fine tuning a new task
  - e.g, take features from network trained for imagenet image classification and just change the last layer for new task



Bonus knob: warm start/pre-training

- Suppose we want to continue training for more epochs
  - save snapshots of weights and resume again
  - need to carefully initialize learning rate now
- Also, can use weights pre-trained from another task as initialization for fine tuning a new task
  - e.g, take features from network trained for imagenet image classification and just change the last layer for new task



Be careful with step size/learning rate

-

Neural Network Optimization

- Main technique: Stochastic Gradient Descent
- Back propagation: allows calculating gradients efficiently
- No guarantees: not convex, can take a long time, but:
  - Often still works fine, finds a good local minimum
- Over parameterization: it *seems* that using LARGE network (sometimes with #weights≫#sameples) helps optimization
  - Remember lecture 2, where doing this was a bad idea!!
  - Not well understood

Optimization

- Check
  - Add gradCheck()
  - Randomly permute data for SGD sequence
- Choose activations to avoid
  - Gradient clipping
  - Gradient explosion
- SGD "knobs" in NN training
  - Initialization → Kaiming/Xavier, or warm start initialization
  - Step size/learning rate → very important to tune based on training/validation loss
  - SGD variants
    - Momentum for SGD → usually added with SGD (default parameter momentum=0.9 often works well)
    - Adaptive variants of SGD → common alternative to SGD+momentum is Adam with $\beta_2 \gg \beta_1, e.g., \beta_2 = 0.999, \beta_1 = 0.9$
  - Mini-batch SGD → ~ 128 common
  - Batch normalization → use batch normalization

Regularization

Using "Too Large" Networks

- It seems that using LARGE network helps optimization.
- Typically, #weight ≈≥ sample size
  - Good generalization even without regularization
  - Not well understood

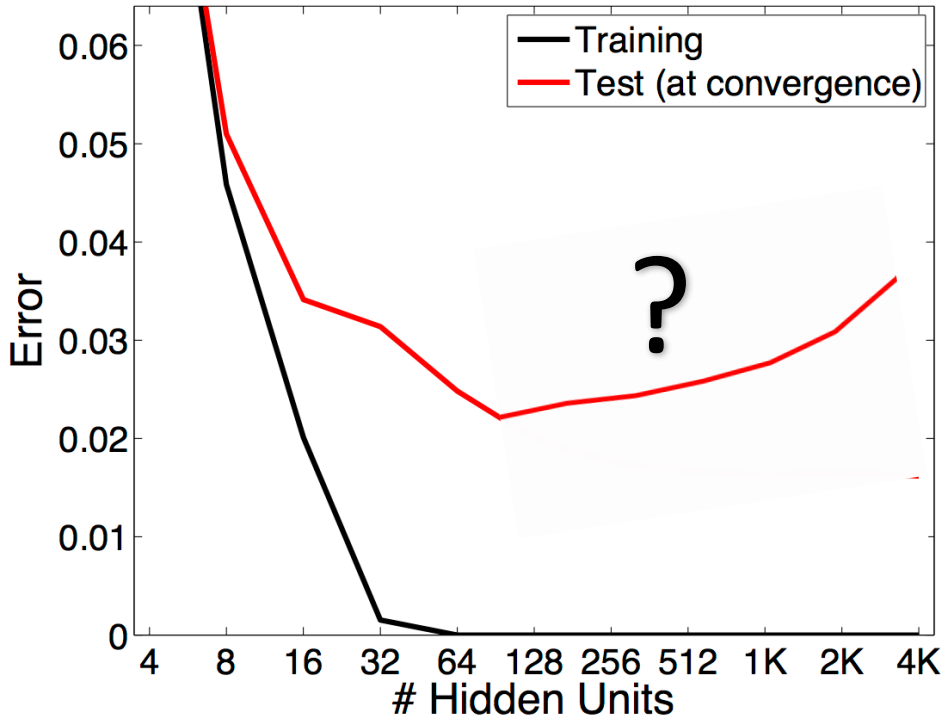
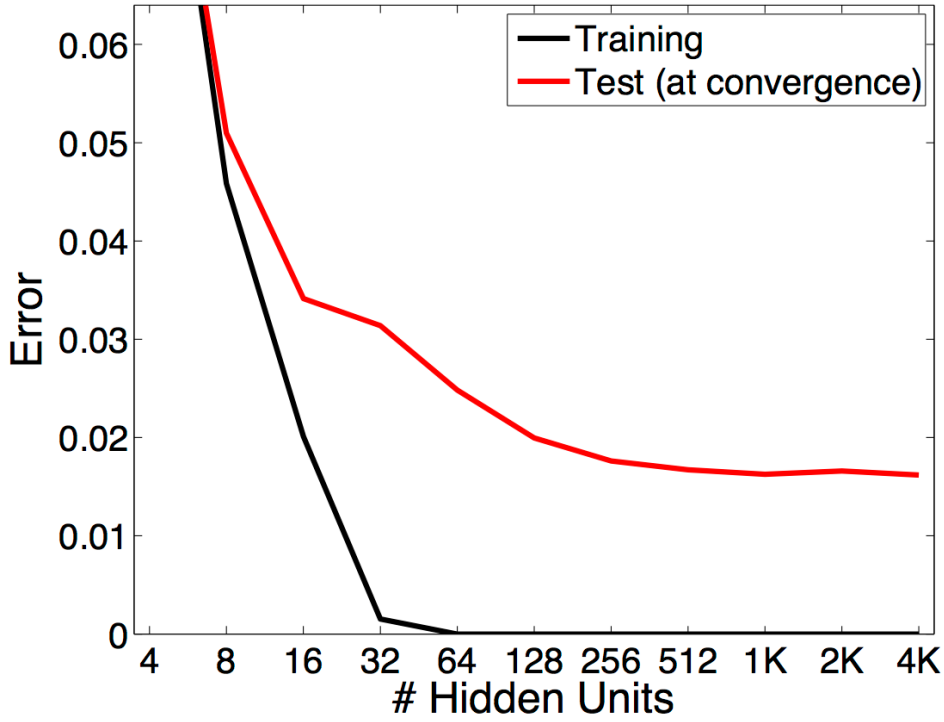


Figure credit: Behnam Neyshabur and Nati Srebro

Using “Too Large” Networks

- It seems that using LARGE network helps optimization.
- Typically, #weight ≈≥ sample size
  - Good generalization even without regularization
  - Not well understood



Figures from Neyshabur et al. 2015

Regularization

- It seems that using LARGE networks helps optimization.
- Typically, #weight $\approx\geq$ sample size
  - Good generalization even without regularization
  - Not well understood
- Still some regularization techniques are commonly used
  - Weight decay
  - Dropout
  - Data augmentation

Regularization - ℓ_2 (weight decay)

- Minimize Regularized ERM

$$\arg\min_{\boldsymbol{W}} L_S(f_{\boldsymbol{W}}) + \frac{\lambda}{2}\|\boldsymbol{W}\|^2$$

- Backpropagation is the same
  - **objective:**

$$\frac{1}{N}\sum_i \left(\ell\left(f_{\boldsymbol{W}}\left(\boldsymbol{x}^{(i)}\right), y^{(i)}\right) + \frac{\lambda}{2}\|\boldsymbol{W}\|^2\right)$$

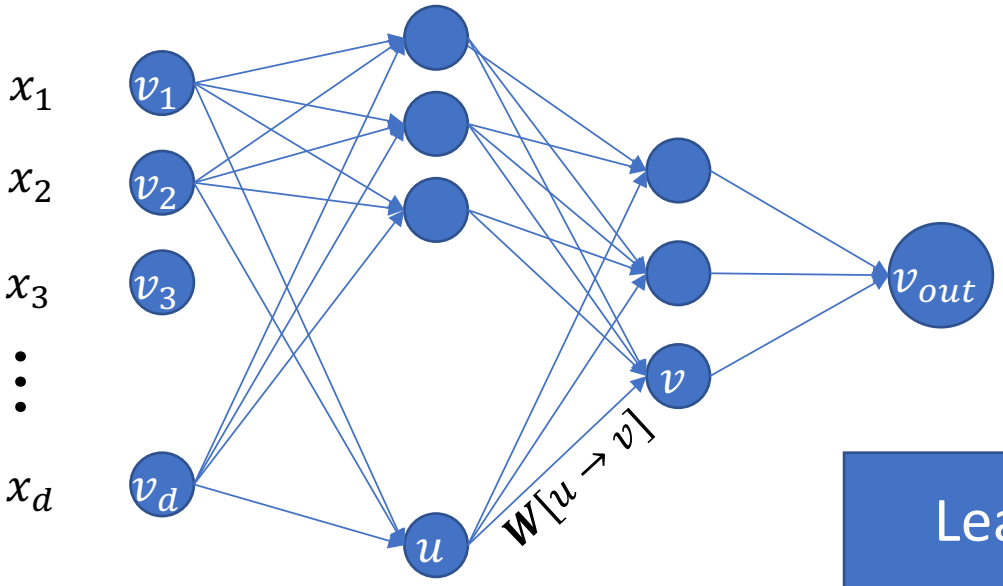
  - **gradient estimate:**

$$\ell\left(f_{\boldsymbol{W}^{(t)}}\left(\boldsymbol{x}^{(i)}\right), y^{(i)}\right) + \lambda\boldsymbol{W}^{(t)} = \boldsymbol{g}^{(t)} + \lambda\boldsymbol{W}^{(t)}$$

  - **updates:**

$$\boldsymbol{W}^{(t+1)} = \boldsymbol{W}^{(t)} - \eta_t \cdot \left(\boldsymbol{g}^{(t)} + \lambda\boldsymbol{W}^{(t)}\right) = (1 - \eta_t\lambda)\boldsymbol{W}^{(t)} - \eta_t\boldsymbol{g}^{(t)}$$

Dropouts



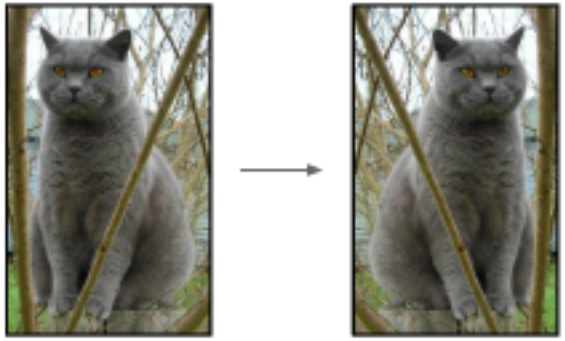
Learn robust representations

- At each step of SGD:
  - Randomly (temporarily) remove p fraction of the units
  - Keep weights between remaining units
  - Update weights between remaining units using backprop (as if removed units don't exist)
- For prediction:
  - Use all units and weights

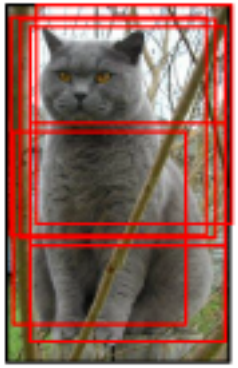
Slide credit: Nati Srebro

Data augmentation

- Augment training data with invariances we know exists for task
  - e.g., image classification
    - translation invariance
    - horizontal invariance
    - rotation invariance (some cases)
    - scale invariance



- Augment training data to have noise/other artifacts in feature space
  - e.g., color jitter, random noise
- Super effective in many computer vision tasks



Figures credit: Andrej Karpathy

Summary

Optimization

- Check
  - Add gradCheck()
  - Randomly permute data for SGD sequence
- Choose activations to avoid
  - Gradient clipping
  - Gradient explosion
- SGD "knobs" in NN training
  - Initialization → Kaiming/Xavier, or warm start initialization
  - Step size/learning rate → very important to tune based on training/validation loss
  - SGD variants
    - Momentum for SGD → usually added with SGD (default parameter momentum=0.9 often works well)
    - Adaptive variants of SGD → common alternative to SGD+momentum is Adam with $\beta_2 \gg \beta_1, e.g., \beta_2 = 0.999, \beta_1 = 0.9$
  - Mini-batch SGD → ~ 128 common
  - Batch normalization → use batch normalization

Regularization

- Data augmentation – very effective
  - Think of what is the right data augmentation for your problem
- Weight decay – tune step sizes/ λ parameter
- Dropout – usually very useful
- Choice of architecture affects validation performance/generalization!
  - why?
- Many optimization choices affect validation performance—unlike convex optimization problems with a unique global minimum, where optimization algorithm only changes the speed/computation of training and not generalization → Not well understood
  - Keep in mind while making choices in previous slides