Deep Learning

- Linear/Logistic regression gradient descent
- Layers:
  - Linear
  - ReLU, Sigmoid
  - Softmax
  - X-ent, X-ent w/ logits
  - Square loss
  - add, mul, sum, max
  - Conv, max-pool
    - padding, stride, dilation
  - batch norm & residual connections
- CNN:
  - general architecture
  - gradients problem & solutions
  - applications
- RNN:
  - simple RNN, $\frac{\partial L}{\partial b}$, $\frac{\partial L}{\partial W}$
  - LSTM & thresholding
  - Applications
- Pretraining
- Optimisation:
  - SGD, why zig-zag, momentum & Nesterov
  - Newton, problem, Gauss Newton
  - Curvature Matrix approximations
- Attention & Memory:
  - Implicit; Jacobian
  - Explicit; glimpse, soft vs hard
- DL for NLP:
  - word embedding
  - LBL/NLM/k-gram/RNN-LM
  - seq-to-seq

DL

- Linear Regression:

$$L(\underline{w}, S) = \frac{1}{2}\sum (y_i - \underline{w}^T \underline{x}_i)^2$$

$$\nabla_{\underline{w}} L = \sum (y_i - \underline{w}^T \underline{x}_i)\underline{x}_i$$

- Logistic Regression:

$$L(\underline{w}, S) = -\sum (y_i \log \sigma(\underline{w}^T \underline{x}_i) + (1-y_i)\log \sigma(-\underline{w}^T \underline{x}))$$

$$\nabla_{\underline{w}} L = -\sum (y_i (1-\sigma(\underline{w}^T \underline{x}_i)) - (1-y_i)\sigma(\underline{w}^T \underline{x}_i))\underline{x}_i$$

$$= \sum (\sigma(\underline{w}^T \underline{x}_i) - y_i)\underline{x}_i$$

- Layer

Layer	$\underline{y}$	$\frac{\partial L}{\partial \underline{x}}$	$\frac{\partial L}{\partial \Theta}$
Linear	$\underline{W}\underline{x} + \underline{b}$	$\frac{\partial L}{\partial \underline{y}} \underline{W}$	$\frac{\partial L}{\partial \underline{W}} = (\frac{\partial L}{\partial \underline{y}})^T \underline{x}^T$ $\quad \frac{\partial L}{\partial \underline{b}} = \frac{\partial L}{\partial \underline{y}}$
ReLU	$y_i = \max(x_i, 0)$	$\frac{\partial L}{\partial x_i} = (y_i > 0)$	
Sigmoid			
Softmax	$y_i = \frac{e^{x_i}}{\sum e^{x_j}}$	$\underline{S} - \sum S_i \quad (S_i = \frac{\partial L}{\partial y_i} y_i)$	
X-ent	$-\sum p_i \log x_i$	$\frac{\partial L}{\partial x_i} = \frac{-p_i}{x_i}$	
(w/ logits)	$-\sum p_i \log(\frac{e^{x_i}}{\sum e^{x_j}})$	$\frac{\partial L}{\partial \underline{x}} = \underline{y} - \underline{p}$	
Square loss	$\lVert \underline{t} - \underline{x} \rVert^2$	$-2\lVert \underline{t} - \underline{x} \rVert$	
Add	$\underline{a} + \underline{b}$	$\frac{\partial L}{\partial \underline{a}} = \frac{\partial L}{\partial \underline{y}}$	
Mul	$\underline{a} \odot \underline{b}$	$\frac{\partial L}{\partial \underline{a}} = \underline{b} \odot \frac{\partial L}{\partial \underline{y}}$	
Sum	$\sum x_i$	$\frac{\partial L}{\partial \underline{x}} = \frac{\partial L}{\partial \underline{y}} \cdot \underline{1}^T$	
Max	$\max x_i$	$\frac{\partial L}{\partial x_i} = \frac{\partial L}{\partial y}$ if $x_i = \max \{\underline{x}\}$ else 0	

- CNN:

$$\text{output} \rightarrow M = \left[\frac{N + 2p - k'}{s}\right] + 1, \quad k' = k + (k-1)(r-1)$$

(input N; dilation r)

s = stride, p = padding

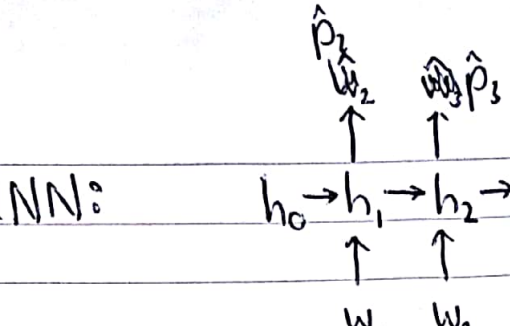
  - SAME padding: $p = (k-1)/2$

[[conv × m] → pool] × n → linear × k → softmax

  - main workhorse: 3×3 conv

- Vanishing / exploding gradients:
  - add batch norm & residual connections
  - careful weight initialisation

- RNN:

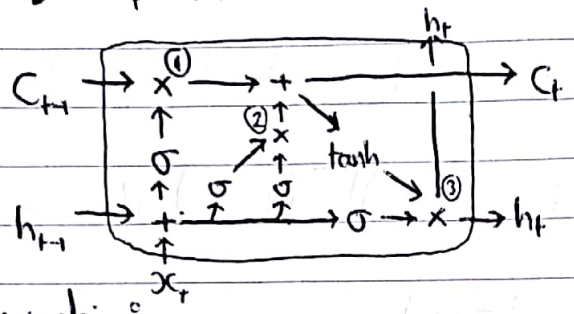


$h_t = \tanh(W h_{t-1} + U w_t)$

$\hat{p}_{t+1} = \text{Softmax}(B h_t)$

$L(W,U,B) = -\sum_t \log p(w_t | w_{1:t-1})$

$\nabla_B L = \sum_t (p_t - \text{label}_t) h_t^T$, $\nabla_W L = \sum_t \sum_{k=1}^{t} \frac{\partial L_t}{\partial p_t} \frac{\partial p_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial W}$



- helps long term dependencies & vanishing / exploding gradients (can also threshold gradients)

- Optimisation:
  - GD: $\Theta_{k+1} = \Theta_k - \alpha_k \nabla_\theta h|_{\theta_k}$
    - implicitly approximate H by $\frac{1}{\alpha} I$; overestimate curvature; zig-zags
  - momentum: $\Theta_{k+1} = \Theta_k + \alpha v_{k+1}$, $v_{k+1} = \eta_k v_k - \alpha_k \nabla_\theta h|_{\theta_k}$
    - fix zig-zagging; Nesterov better theoretical bound, similar in practice
  - 2nd order:
    - Newton: $\Theta_{k+1} = \Theta_k - \alpha H^{-1} \nabla_\theta h|_{\theta_k}$
      - H under-estimates curvature outside small radius
      - can limit to r; replace H with $(H + \lambda I)$
      - or use other matrix, eg $G = \sum J_i^T H_i J_i$, J_i = Jacobian of $f(x,\theta)$ wrt θ, H_i = Hessian of $l(y,\hat{y})$ wrt $\hat{y}$
        - G works better in practice (& always PSD)
    - Curvature matrix approximations:
      - $H: D \times D$, H^{-1} $O(D^3)$ (& $O(D^2)$ storage)
      - too expensive; use approximations to speed up
      - diagonal of empirical Fisher matrix (RMSProp, Adam)
      - low rank approx (L-BFGS)
    - No asymptotic improvement from stochastic 2nd order, but improvement before asymptote ⇒ can still be good to minibatch
- Attention & Memory (RNN):
  - Implicitly given by Jacobian; $J_{ij} = \frac{\partial y_i}{\partial x_j}$
  - Explicitly done by attention model using network output + extra data to make a glimpse; then fed back into network as input
  - Hard glimpse not differentiable, train by RL. Soft-glimpse is; GD works good!

COMPGI13
Advanced Topics in Machine Learning

Deep Learning

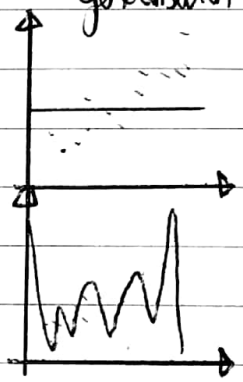
Supervised Learning

$\underline{x} \rightarrow \boxed{f} \rightarrow y$ $\begin{cases} y \in \{-1,+1\}, \text{classification} \\ y \in \mathbb{R}, \text{regression} \end{cases}$

- Generalisation:
  - performance on unseen data
  - empirical risk minimization: minimize training error (biased!)
  - validation: hold out data to get unbiased estimator (eg cross validation)

$$\underbrace{\text{validation error}}_{\text{what we care about}} = \text{training error} + \underbrace{(\text{validation error} - \text{training error})}_{\text{generalisation error}}$$

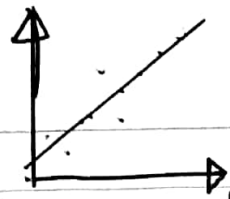
- Underfitting:
  - error driven by approximation
  - high bias, low variance
- Overfitting:
  - error driven by generalisation
  - low bias, high variance



- Early stopping:
  - control degree of overfitting in iterative optimisation methods
- Regularisation:
  - impose a preference for simpler hypothesis
  - weight decay, (L1 or L2), dropout..
- Gradient descent: $\underline{w} \leftarrow \underline{w} - \eta \nabla L(\underline{w})$ (loss function to minimise)
  - useful when no closed form solution / dataset too large/ high dimensional to practically solve
  - often practical to use a mini-batch S_j to approximate $\nabla L(\underline{w})$, where $L(\underline{w}) = \sum_i f_i(\underline{w})$

$$\nabla L(\underline{w}) \approx \sum_{i \in S_j} \nabla f_i(\underline{w})$$

- minibatch size trades off computational cost & variance



Squared error = p Gaussian noise

- Linear Regression:

$\hat{y}(\underline{w}, \underline{x}) = \underline{w}^T \underline{x}$, $l(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2$

$L(\underline{w}, S) = \frac{1}{2}\sum_{i=1}^{m} (y_i - \hat{y}(\underline{w}, \underline{x}_i))^2$

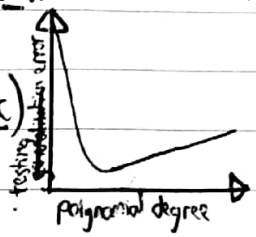
~~$\nabla l = y - \hat{y}$~~ $\nabla_{\underline{w}} L(\underline{w}, S) = \sum_i (y_i - \underline{w}^T \underline{x}_i)\underline{x}_i$

regularisation → · weight decay: $L'(\underline{w}, S) = L(\underline{w}, S) + \lambda \|\underline{w}\|^2$ ← Gaussian prior on weights

· Non-linear basis functions:

· $\phi(\underline{x}) = (1, x, x^2)$, $\hat{y}(\underline{w}, \underline{x}) = \underline{w}^T \phi(\underline{x})$

· provides richer hypothesis space



- Logistic Regression: $y \in \{0, 1\}$

$z(\underline{w}, \underline{x}) = \underline{w}^T \underline{x}$, $\hat{p}(z) = \frac{1}{1 + e^{-z}}$

cross entropy loss → $l(y, \hat{p}) = y \log \hat{p} + (1 - y) \log (1 - \hat{p})$

$L_{\underline{w}}(\underline{w}, S) = -\sum_i l(y, \hat{p})$

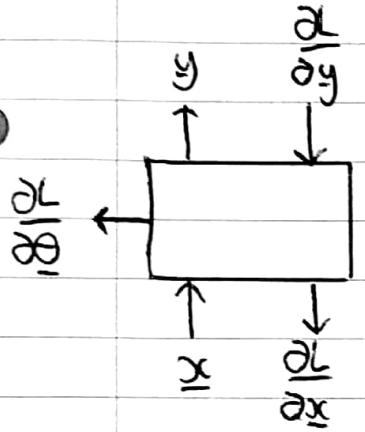
$= \sum_i \log(1 + e^{\underline{w}^T \underline{x}_i}) - y_i \underline{w}^T \underline{x}_i$

$\nabla_{\underline{w}} L(\underline{w}, S) = \sum_i \frac{x_i e^{\underline{w}^T \underline{x}_i}}{1 + e^{\underline{w}^T \underline{x}_i}} - y_i \underline{x}_i$

$= \sum_i \left(\frac{1}{1 + e^{-\underline{w}^T \underline{x}_i}} - y_i \right) \underline{x}_i$

Neural Networks: Foundations

- Composition of linear transforms + non-linear functions
  - learn by chain rule derivatives & SGD on loss function
  - each node in compute graph has:
    - forward_pass: $\underline{x} \to \underline{y}$ (computing output & loss)
    - backward_pass: $\frac{\partial L}{\partial \underline{y}} \to \frac{\partial L}{\partial \underline{x}}$ (backpropagation)
    - compute_gradients: $\underline{x}, \frac{\partial L}{\partial \underline{y}} \to \frac{\partial L}{\partial \theta}$ (for param updates)
- Module zoo:



- Linear layer: (affine linear transform)

$$\underline{y} = \underline{\underline{W}}\,\underline{x} + \underline{b}$$

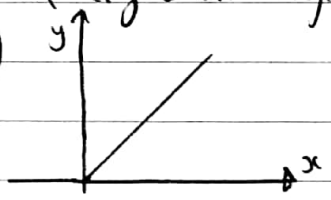
$$\frac{\partial L}{\partial \underline{x}} = \frac{\partial L}{\partial \underline{y}}\,\underline{\underline{W}}$$

$$\frac{\partial L}{\partial \underline{\underline{W}}} = \left(\frac{\partial L}{\partial \underline{y}}\right)^T \underline{x}^T, \quad \frac{\partial L}{\partial \underline{b}} = \frac{\partial L}{\partial \underline{y}}$$

- ReLU: (rectified linear layer; non-linearity)

$$y_i = \max(x_i, 0)$$

$$\frac{\partial L}{\partial x_i} = (y_i > 0)$$



- Softmax:

$$y_i = \frac{e^{x_i}}{\sum e^{x_j}}$$

$$\frac{\partial L}{\partial \underline{x}} = \underline{S} - \sum_i S_i, \quad S_i = \frac{\partial L}{\partial y_i}\, y_i$$

- Cross-entropy loss: (p = label probabilities)

$$y = -\sum_i p_i \log x_i$$

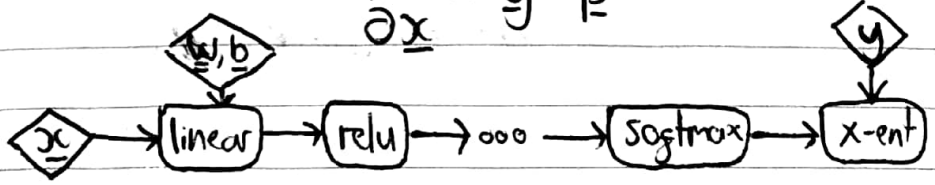
$$\frac{\partial L}{\partial x_i} = \frac{-p_i}{x_i}$$

good to pair with softmax ↓ (softmax + xent loss in 1)

- Cross-entropy loss w/ logits:

$$y = -\sum_i p_i \log\left(\frac{e^{x_i}}{\sum_j e^{x_j}}\right)$$

$$\frac{\partial L}{\partial \underline{x}} = \underline{y} - \underline{p}$$



element wise

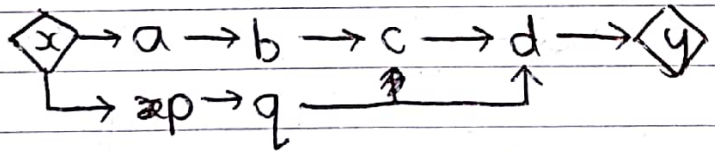
- Add: $y = \underline{a} + \underline{b}$ — $\frac{\partial L}{\partial \underline{a}} = \frac{\partial L}{\partial y}$
- Mul: $y = \underline{a} \odot \underline{b}$ — $\frac{\partial L}{\partial \underline{a}} = \underline{b} \odot \frac{\partial L}{\partial y}$

group wise

- Sum: $y = \sum x_i$ — $\frac{\partial L}{\partial \underline{x}} = \frac{\partial L}{\partial y} \underline{1}^T$
- Max: $y = \max(x_i)$ — $\frac{\partial L}{\partial x_i} = \frac{\partial L}{\partial y}$ if x_i max, else 0

- Leaky-ReLU: addresses problem of 0-gradient on ReLU for $x_i < 0$ (calls asleep)
- Squared loss: $y = \|\underline{t} - \underline{x}\|^2$ — $\frac{\partial L}{\partial \underline{x}} = -2(\underline{t} - \underline{x})^T$

- Optimisation: (backtrack thru forward execution pass)
  - Forward pass to find loss
  - Backward pass to find gradients w.r.t. inputs
  - Compute gradients w.r.t. parameters & update by gradient descent



$$y = d(c(b(a(x)), q(p(x))))$$

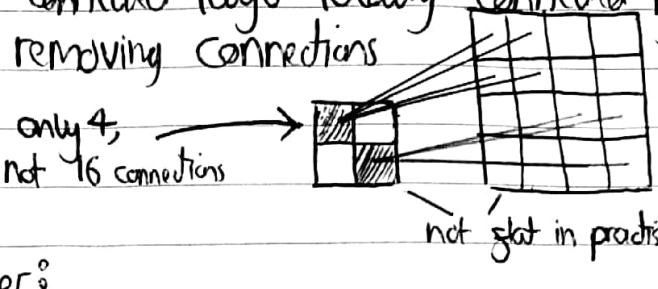
$$\frac{\partial y}{\partial x} = \frac{\partial a}{\partial x}\cdot\frac{\partial b}{\partial a}\cdot\frac{\partial c}{\partial b}\cdot\frac{\partial d}{\partial c}\cdot\frac{\partial y}{\partial d} + \frac{\partial p}{\partial x}\frac{\partial q}{\partial p}\frac{\partial d}{\partial q}\frac{\partial y}{\partial d}$$

(backprop)

- Issues:
  - overfitting: regularisation (weight decay, dropout)
  - initialisation: scaling, batch-norm
  - hyperparameters: random vs grid search

Image Recognition with Convolutional Networks

- Aim for translational invariance
  - Make fully connected layer locally connected by sharing weights & removing connections



- Convolutional layer:

$$y = \sum_{i \in \text{Receptive field}} w_i x_i + b = \underline{w} * \underline{x} + \underline{b}$$

('slide' receptive field over image & compute dot products)

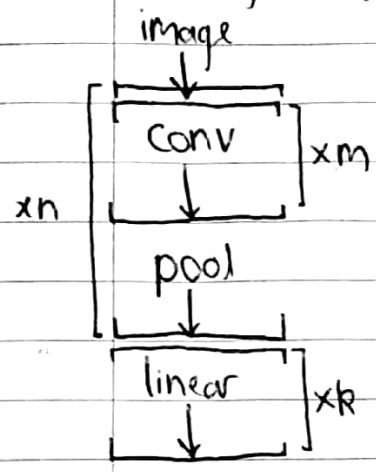
  - for $N \times N$ input & $k \times k$ kernel, output is $M \times M$, where $M = N - k + 1$
  - Padding:
    - VALID: no padding
    - SAME: pad with $p = (k-1)/2$ zeros on all sides, so output same size as input
  - Stride:
    - filter takes steps larger than 1
    - reduce computational cost / spatial resolution, & invariance to local translation
  - Dilation:
    - rate r = enlarge receptive field but keep same num parameters
    - $k' = k + (k-1)(r-1)$

$$M = \left[\frac{N + 2p - k'}{s}\right] + 1$$

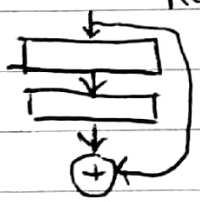
, for input $N \times N$, p padding, k' dilated kernel size & s stride length

  - Deconvolution:
    - swap direction of convolution; increases resolution
    - used in backprop
- Pooling layer:
  - reduces spatial resolution
  - average, or max pooling over receptive field
  - can do global pooling too; full invariance to location

- Building deep networks:
  - usually follow layers with a Relu
  - deeper usually better; more invariances & non-linearities
  - typically build up no. channels over depth, before linear, FC layers
  - main workhorse: 3×3 convolution layers
    - stacking these up gives effective larger receptive fields



- Training:
  - hard to train very deep nets; vanishing or exploding gradients
  - Fixes:
    - Weight initialisation:
      - 0 mean Gaussian low variance, or, eg, variance to preserve gradient magnitude
    - Batch normalisation:
      - layer in network
      - normalise input to zero mean, unit variance
    - Residual connections:
      - very deep networks get higher training error, as hard to learn identity function
      - add connections which skip layers
      - also improves gradient flow, as can skip layers



Sequences & Recurrent Neural Nets

$h_0 \to h_1 \to h_2 \to \cdots$ (with w_1, w_2, w_3)

$$\underline{h}_t = \tanh(\underline{\underline{W}}\,\underline{h}_{t-1} + \underline{\underline{U}}\,\underline{w}_t)$$

$$p(\underline{w}_{t+1}) = \text{softmax}(\underline{\underline{B}}\,\underline{h}_t)$$

$$L(\underline{\underline{W}}, \underline{\underline{U}}, \underline{\underline{B}}) = \sum_{t=1}^{T} L_t,$$

$$L_t = -\log p(w_t \mid w_{1:t-1})$$

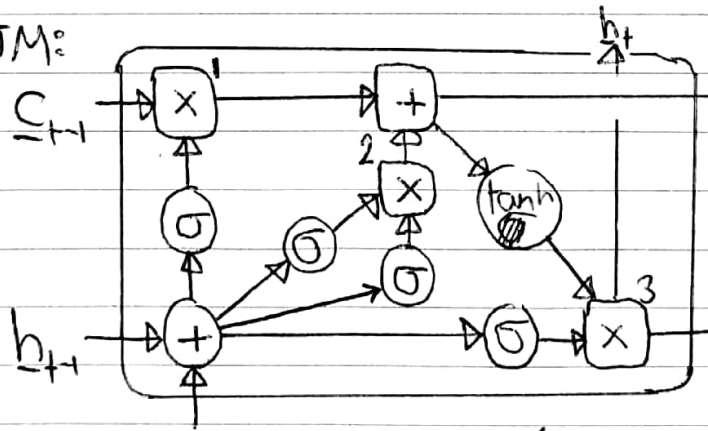
$$\frac{\partial L_t}{\partial \underline{\underline{B}}} = \frac{\partial L_t}{\partial p_t}\frac{\partial p_t}{\partial h_t}\frac{\partial h_t}{\partial \underline{\underline{B}}}$$

$$= (p_t - \text{label}_t)\,h_t^T \Rightarrow \frac{\partial L}{\partial B} = \sum_{t=1}^{T}\frac{\partial L_t}{\partial \underline{\underline{B}}}$$

$$\frac{\partial L_t}{\partial \underline{\underline{W}}} = \frac{\partial L_t}{\partial p_t}\frac{\partial p_t}{\partial h_t}\frac{\partial h_t}{\partial \underline{\underline{W}}} = \sum_{k=1}^{t}\frac{\partial L_t}{\partial p_t}\frac{\partial p_t}{\partial h_t}\frac{\partial h_t}{\partial h_k}\frac{\partial h_k}{\partial \underline{\underline{W}}}$$

- problem: $\underline{h}_t = \underline{\underline{W}}^T \underline{h}_{t-1}$, $\underline{h}_t \to \infty$ if $|\underline{\underline{W}}| > 1$; $\underline{h}_t \to \underline{0}$ if $|\underline{\underline{W}}| < 1$
  (vanishing or exploding gradients)
  - makes learning long term dependencies very hard

- LSTM:

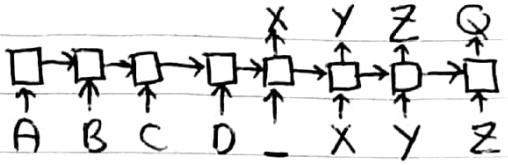


  - cell state C_t allows for long term dependencies
  - 1: forget gate
    2: input gate
    3: output gate

(each σ or tanh is linear layer (w/ bias) followed by activation function)

- Can also help solve exploding gradients by thresholding:
  if $\|g\| > \text{threshold}$:
  $$g \leftarrow \frac{\text{threshold}}{\|g\|} \times g$$

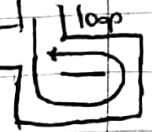
- Applications:
  - Modelling languages; sequence of words
  - Pixel prediction; image as sequence
  - Sequence to sequence:
    - image captioning, speech
    - Machine translation



Beyond Image Recognition: End-to-End Learning, Embeddings

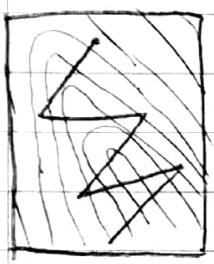
- Pretraining:
  - use a ready trained network, and use as base to build on
  - don't update these params; just feed output into new network (or update slowly; low learning rate)
  - eg pre-using network pre-trained on ImageNet for other image datasets

- Object detection:
  - As classification; classify all boxes (may be repeated objects; slow)
  - As prediction; regress 4 corners of box; unknown no. boxes
  - Bounding box proposal:
    - module predicts which boxes may be objects
    - then passes each box to a classifier
- Semantic Segmentation:
  - label each pixel into an object class
  - output full resolution map with 1 channel per class & probabilities for each pixel
    - 'fully convolutional networks'; no linear layers in whole network
- Spatial transformers:
  - learn to transform input, eg rotate, to some canonical representation
- Similarity / dissimilarity
  - inputs associated with each other; eg all images of a given persons face
  - use different loss functions based on predicted similarity
- Maze navigation:
  - problem: sparse rewards; we want some form of spatial knowledge;
  - add auxillary rewards for auxillary tasks; depth prediction or loop closure prediction



Optimisation for Machine Learning

- Gradient Descent: $\underline{\theta}_{k+1} = \underline{\theta}_k - \alpha_k \nabla h(\underline{\theta}_k)$ (objective, parameters; $\nabla h(\underline{\theta}_k)$: greatest reduction in h)
  - Motivated by 1st order Taylor expansion
    $$h(\underline{\theta}+\underline{\delta}) \approx h(\underline{\theta}) + \nabla h(\underline{\theta})^T \underline{\delta}$$
  - reasonable for small δ, ie low learning rates
  - problem: oscillates along a valley
    - cannot solve by fine tuning learning rate; either big zig-zag or slow progress
  - reason: we minimize local quadratic approximation to $h(\theta)$:
    $$h(\underline{\theta}+\underline{\delta}) \approx h(\underline{\theta}) + \nabla h(\underline{\theta})^T \underline{\delta} + \frac{1}{2}\underline{\delta}^T H(\underline{\theta})\underline{\delta}$$
    - approximate $H(\underline{\theta})$ with $L\underline{\underline{I}}$; treat all directions as having same curvature, $\frac{1}{2}\underline{\delta}^T(L\underline{\underline{I}})\underline{\delta} = \|\underline{\delta}\|^2 \cdot \frac{L}{2}$
    $$h(\underline{\theta}+\underline{\delta}) \approx h(\underline{\theta}) + \nabla h(\underline{\theta})^T \underline{\delta} + \frac{L}{2}\|\underline{\delta}\|^2$$
    $$\Rightarrow -\frac{1}{L}\nabla h(\underline{\theta}) = \operatorname{argmin}_{\underline{\delta}} \left(h(\underline{\theta}) + \nabla h(\underline{\theta})^T \underline{\delta} + \frac{L}{2}\|\underline{\delta}\|^2 \right)$$
    - $\Rightarrow$ struggle when curvature varies a lot, ie $H(\underline{\theta}) \neq L\underline{\underline{I}}$



Asymptotic performance; often stop before this point anyway

Worst case; very pessimistic

- Convergence theory:
  $$h(\underline{\theta}_k) - h(\underline{\theta}^*) \leq \frac{L}{2}\left(\frac{\kappa-1}{\kappa+1}\right)^{2k} \|\underline{\theta}_0 - \underline{\theta}^*\|^2 \quad \left(\text{for } \alpha_k = \frac{2}{L+\mu}\right.$$

worst case upper bound

$\Rightarrow$ iterations for $h(\underline{\theta}_k) - h(\underline{\theta}^*) \leq \epsilon$: $k \in O\left(\kappa \log \frac{1}{\epsilon}\right)$ $\kappa = \frac{L}{\mu}$

- Momentum:
  $$\underline{v}_{k+1} = \eta_k \underline{v}_k - \alpha_k \nabla h(\underline{\theta}_k) \qquad \eta_k$$: friction constant
  $$\underline{\theta}_{k+1} = \underline{\theta}_k + \underline{v}_{k+1} \qquad \alpha_k$$: learning rate
  - helps to fix zig-zagging problem by updating w/ moving average of gradient
  - Convergence theory:
    - another first order method (update is linear combination of previous gradient estimates)
    - ~~Now~~ $k \in O\left(\sqrt{\kappa} \log \frac{1}{\epsilon}\right)$; $\sqrt{\kappa}$ factor speedup

worst case lower bound

↖ upper bound for Nesterov, but similar performance in practice

- Second order methods:
  - Approximate $h(\underline{\theta}+\underline{\delta})$ w/ 2nd order Taylor expansion

$$h(\underline{\theta}+\underline{\delta}) \approx h(\underline{\theta}) + \nabla h(\underline{\theta})^T \underline{\delta} + \frac{1}{2}\underline{\delta}^T \underline{\underline{H}}\,\underline{\delta}$$

$$\Rightarrow \underline{\underline{H}}^{-1}\nabla h = \arg\min_{\delta}\left(h(\underline{\theta}) + \nabla h(\underline{\theta})^T\underline{\delta} + \frac{1}{2}\underline{\delta}^T\underline{\underline{H}}\,\underline{\delta}\right)$$

$$\underline{\theta}_{k+1} = \underline{\theta}_k - \underline{\underline{H}}^{-1}\nabla h(\underline{\theta}_k)$$

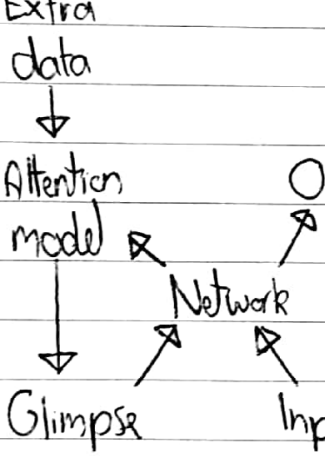
  - Problems:
    - approximation only valid near θ_k
    - 1st order methods approximate $\underline{\underline{H}}$ w/ $L\underline{\underline{I}}$, here we ~~approximate~~ use real $H(\underline{\theta})$, but this may underestimate curvature as we move away from $\underline{\theta}$
    - Solution: constrain $\underline{\delta}$ to be in some region R, near $\underline{\theta}$ where our approximation remains good
      - if $R = \{\underline{\delta} : \|\underline{\delta}\|^2 \leq r\}$, then doing our $\arg\min_{\underline{\delta}\in R}(\cdots)$ is ~~approxi~~ equivalent to doing $\arg\min_{\underline{\delta}}(\cdots)$, but replacing $H(\underline{\theta})$ with $(H(\underline{\theta}) + \lambda\underline{\underline{I}})$, where λ is some function of r
      - can also instead replace $\underline{\underline{H}}$ with another matrix, upper bounding curvature in ~~over~~ larger regions
        - eg Generalized Gauss Newton Matrix
          Fisher information matrix
          Empirical fisher info matrix
- Generalized Gauss Newton:
  - assume $h(\underline{\theta}) = \sum h_i(\underline{\theta}) = \sum l(y_i, f(x_i, \underline{\theta}))$
  - $\underline{\underline{G}} = \sum \underline{\underline{J}}_i^T \underline{\underline{H}}_i \underline{\underline{J}}_i$, $\underline{\underline{J}}_i$ is Jacobian of $f(x_i, \underline{\theta})$ wrt $\underline{\theta}$
    $\underline{\underline{H}}_i$ is Hessian of $l(y_i, z_i)$ wrt z_i
  - $\underline{\underline{G}} = \underline{\underline{H}}$ if replace $f(x_i, \underline{\theta})$ w/ 1st order approximations
  - for squared loss, $\underline{\underline{H}}_i = \underline{\underline{I}} \Rightarrow \underline{\underline{G}} = \sum \underline{\underline{J}}_i^T \underline{\underline{J}}_i$
  - $\underline{\underline{G}}$ always PSD, more conservative than Hessian & works much better in practise

Optimisation 2

- Curvature matrix approximations:
  - θ can have 10s of millions of dimensions
  - $\underline{\underline{H}}$ (or some other curvature matrix $\underline{\underline{B}}$) is $n \times n$ & $O(n^3)$ to invert; impractical
  - Approximations:
    - diagonal: $O(n)$ storage, $O(n)$ to invert
      - only reasonable if eigenvalues of $\underline{\underline{B}}$ closely alligned w/ coordinate axes
      - popular is empirical Fisher (used by RMS-prop & Adam)
    - low-rank: $O(rn)$ storage, computation & to inverse
      - diagonal + rank r correlations
      - best if $\underline{\underline{B}}$ has few important eigenvectors w/ large eigenvalues
      - eg L-BFGS
    - block-diagonal: $O(bn)$ to store, $O(b^2 n)$ to inverse
      - can be awkward to compute (similar to diagonal)
      - eg TONGA
    - Kronecker-product: $O(n)$ to store/compute, $O(\sqrt{B}n)$ to inverse
      - block diagonal of GGN / Fisher
      - blocks correspond to layers, each approximated by Kronecker-product of 2 smaller matrices
- Stochastic optimisation:
  - randomly sample mini-batch to compute gradient
  - useful for momentum, but must be careful w/ α & η choice
  - for second order methods*, gives no asymptotic improvement (*and momentum)
    - but pre-asymptotically can help, which is what we care most about

Attention & Memory in Deep Learning

- Deep networks have implicit attention; respond more to certain parts of input than to others (shown by Jacobian)
  $J_{ij} = \frac{\partial y_i}{\partial x_j}$ (compute w/ backprop; set loss to output activation)
- In RNNs, attention ~~gives~~ shows ~~a form of~~ memory; attention to past inputs to produce present output
  - shown by sequential Jacobian; $\underline{J}_k^t = \left(\frac{\partial y_k^t}{\partial \underline{x}^1}, \frac{\partial y_k^t}{\partial \underline{x}^2}, \cdots\right)$
  - eg delayed dot of 1 in handwriting, or out-of-order machine translation
- Explicit attention:
  - limit data presented to network in some way
  + ↑ computational efficiency, scalability, interpretability



- attention model chooses what ~~to~~ should be visible to network in next time step; the 'glimpse'
- whole system recurrent, even if network isn't
- Glimpse: $P(g|a)$
  - generally define probability distribution over glimpses of data, given some set of attention outputs from the network

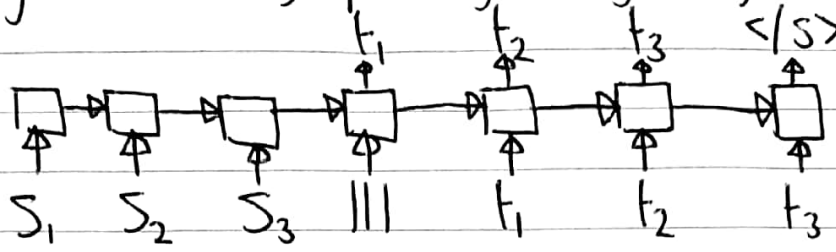
not differentiable → hard attention; specific window passed to network

- can train this w/ RL techniques; reward = loss induced by the glimpse (can use RL whenever a module not differentiable)
- glimpse could eg be Gaussian w/ given mean & variance, over coordinates of an image, defining a window to pass to network
- Generally prefer soft attention; differentiable, can train end-to-end with backprop!
  - take expectation from $P(g|a)$ rather than a sample

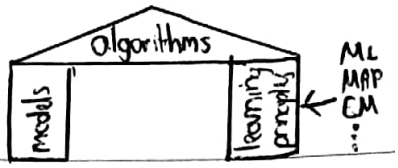
Deep Learning for NLP

- Word Vectors:
  - sparse, orthogonal & semantically weak
  - we want dense representation based on context/use of words
  - Count-based:
    - C = context words
    - count words in C in window around a given word to determine it's vector representation
    - need to normalise counts; distinguish informative high counts from uninformative (eg tf-idf, pmi...)
    - measure similarity by cosine similarity $\left(\text{cosine}(\underline{u},\underline{v}) = \frac{\underline{u}\cdot\underline{v}}{\|\underline{u}\|\,\|\underline{v}\|}\right)$
  - Neural embedding models:
    - find $\underset{\underline{\Theta},\underline{E}}{\text{argmin}}\ L(\underline{\Theta},\underline{E})$, $L(\underline{\Theta},\underline{E}) = -\sum_{t}\sum_{t_i} \text{score}(t_i, c(t_i), \underline{\Theta}, \underline{E})$
      - t is a word in vocab
      - t_i is an instance of t in data
      - $\underline{E}$ is embedding
      - score function important;
        - must embed t_i with $\underline{E}$, be differentiable & take $c(t_i)$ (context) into account
- Discrete Language Models:
  - Goal: estimate $P(S_{1:n}) \approx \prod_{i=1}^{n} P(S_i \mid S_{1:i-1})$
  - $S_{1:i-1}$ variable length; use n-gram (fixed window size)
  - too small n = bad model, too large = counts too low; suffer from sparsity
- Continuous Condition Language Models:
  - LBL/NLM:
    - co-learn word embeddings, then use n-gram model on them
    - log-bilinear model = linear from embeddings to probabilities
    - neural language model = sigmoid activation function
  - RNN-LMs:
    - no longer need k-Markov assumption ((in practice still some soft limitations on k based on data, memory size etc)

- Sequence-to-sequence mapping w/ RNNs:
  - eg translation, parsing (string→tree), or more general computation



  - train to maximise likelihood of $\underline{t}$ given $\underline{s}$
  - problem:
    - non-adaptive capacity, & modelling target sentence dominates training
    - encoder 'gradient starved' ← (attention models can help here)
- Semantic Composition:
  - language free structured; get parse tree distribution & exploit this with an RNN
  - ~~can, eg, annotate~~

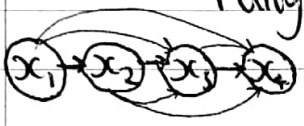


Unsupervised Learning & Generative Models

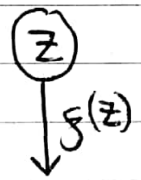
- Probalistic Machine Learning:
  - Model types:
    - Unconditional: $p(x)$, no target / labels
      - ~~the~~ what generative model usually means
    - Conditional $p(y|x)$
      - eg supervised learning; regression & classification

    ] all are actually generative models

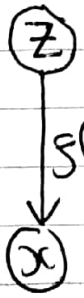
  - Types of generative models:
    - Fully observed models:



      - model data directly; don't introduce any unobserved variables
      - directed graphical models:
        - \+ simple to learn parameters (log-likelihood computable)
        - \+ scales well
        - − order sensitive
      - undirected hard to learn parameters
      - − slow to ~~sample~~ generate; need to go thru elements sequentially
    - Implicit models:

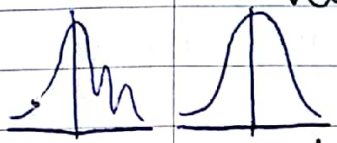


      - transform unobserved noise source using parameterised function
      - \+ easy to sample & compute expectations
      - − hard to maintain invertibility & to optimize
      - − lack of a noise model
    - Latent variable models
      - local random variables representing hidden causes



      - \+ easy to sample
      - \+ encode hierarchy, depth, structure
      - \+ ~~latents~~ compress & represent data
      - \+ can score, compare & select models by likelihood
      - − ~~to~~ inversion difficult
      - − Merginal likelihood requires approximations

- Inference in Prescribed Probabilistic Models:
  - Model evidence:
    $$p(\underline{x}) = \int p(\underline{x}, \underline{z})\, d\underline{z}$$
    - aim to improve this for given data sets
    - intractable in general; transform into expectation over known distribution
  - Jensen's inequality:
    - for concave functions ξ,
      $$\xi(E(x)) \geq E(\xi(x))$$
  - Variational Inference



$$p(\underline{x}) = \int p(x|z)p(z)\,dz$$
$$p(x) = \int p(x|z)\frac{p(z)}{q(z)}q(z)\,dz$$
Jensens inequality
$$\log p(x) \geq \int q(z)\log\left(p(x|z)\frac{p(z)}{q(z)}\right)dz$$
$$= \int q(z)\log p(x|z) - \int q(z)\log\frac{p(z)}{q(z)}\,dz$$
Variational lower bound
$$= E_{q(z)}[\log p(x|z)] - KL[q(z)\|p(z)]$$

- $q(z)$ is our approximation to $p(z)$, using some known, simple distribution
- $KL(q\|p)$ is a 'penalty' term, as $q \neq p$
- optimise for q; eg is Gaussian, variational parameters are mean & variance to optimise for
  - this can tighten the bound, getting closer to true marginal likelihood

max likelihood for other

- Only some parts of model given prob distributions, eg latent vars (in Variational Bayes, all unknown quantities are probability distributions)

+ applicable to all probabilistic models
+ convert integral problem to optimisation
+ numerically stable & parallelisable, fast to converge
± need to choose form of q, and only approximate
− local minima
− can under estimate variance in posterior

Unsupervised Learning & Generative Models

- Families of Approximate Posterior distributions:
  - Free form:
    - get $q(\underline{z}) \propto p(\underline{z}) e^{\log p(\underline{x}|\underline{z},\theta)}$
    - problem: solving is normalisation of original problem
  - Fixed form:
    - specify explicit form of q; $q_\phi(\underline{z}) = f(\underline{z};\phi)$
    - Mean field methods:
      - assume distribution is factorised
      - can introduce structure to factorisation (dependencies) to be more expressive
      - eg autoregressive approximation: $q(\underline{z}) = \prod_i q_i(z_i|z_{1:i})$ (condition on all previous variables)
    - Variational Latent Gaussian models:
      - $\underline{z} \sim N(\underline{z}|0,1)$, $\underline{y} \sim P(\underline{y}|f_\theta(\underline{z}))$, $q(\underline{z}) = \prod_i N(z_i|\mu_i, \sigma_i^2)$

$$F(\underline{y}, q) = E_{q(\underline{z})}[\log p(\underline{y}|\underline{z})] - KL[q(\underline{z})||p(\underline{z})]$$

$$= E_{q(\underline{z})}[\log p(\underline{y}|\underline{z})] - \sum_i KL\left[N(z_i|\mu_i,\sigma_i^2), N(z_i|0,1)\right]$$

$$= E_{q(\underline{z})}[\log p(\underline{y}|\underline{z})] - \frac{1}{2}\sum_i [\sigma_i^2 + \mu_i^2 - 1 - \ln \sigma_i^2]$$

- Variational Optimisation:
  - EM algorithm
  - E Step: $\phi \propto \nabla_\phi F(\underline{x}, q)$ ← (variational params)
    - $\phi_n \propto \nabla_\phi E_{q_\phi(\underline{z})}[\log p_\theta(\underline{x}_n|\underline{z}_n)] - \nabla_\phi KL[q(\underline{z}_n)||p(\underline{z}_n)]$
  - M step: $\theta \propto \nabla_\theta F(\underline{x}, q)$ ← (model params)
    - $\theta \propto \frac{1}{N}\sum_n E_{q_\phi(\underline{z})}[\nabla_\theta \log p_\theta(\underline{x}_n|\underline{z}_n)]$
  - Can do stochastic approach; do EM on minibatches
    - More scalable
  - If expected log likelihood & KL-divergence can't be calculated, use Monte-Carlo methods
    - doubly stochastic estimation; stochasticity from minibatch & Monte-Carlo estimation

- Learning in Implicit Probabilistic Models:
  - Estimation by comparison:
    - compare estimate $q(x)$ with true $p^*(x)$

      $\frac{p^*(x)}{q(x)} = 1 \qquad p^*(x) = q(x)$

    - test a hypothesis, then modify q slightly to better match p^* result
  - Adversarial Learning:

Comparison loss $\rightarrow \Theta \propto \nabla_\theta E_{p^*(x)}[\log D_\theta(x)] + \nabla_\theta E_{q_\phi(x)}[\log(1 - D(x)]$

Generative loss $\rightarrow \phi \propto -\nabla_\phi E_{q(z)}[\log(1 - D_\theta(\xi_\phi(z)))]$

$$\left(\frac{p^*(x)}{q(x)} = \frac{p(x|y=1)}{p(x|y=-1)} = \frac{p(y=1|x)}{p(y=-1|x)} = \frac{D_\theta(x)}{1 - D_\theta(x)} \right)$$