Deep Learning-2

Learning Deep Networks

- Finding the parameters θ of a neural network that significantly reduce a cost function J (θ), which typically includes a performance measure evaluated on the entire training set as well as additional regularization terms.
- The cost function can be written as an average over the training set, such as

$$J(\boldsymbol{\theta}) = \mathbb{E}_{(\boldsymbol{x},\mathrm{y})\sim\hat{p}_{\mathrm{data}}} L(f(\boldsymbol{x};\boldsymbol{\theta}), y),$$

- where L is the per-example loss function, f (x; θ) is the predicted output when the input is x, ˆpdata is the empirical distribution.
  - In the supervised learning case y is the target output.

Objective Function

- Objective function is typically with respect to the training set
  - We would usually prefer to minimize the corresponding objective function where the expectation is taken across the data generating distribution p-data rather than just over the finite training set
- Empirical Risk Minimization

$$\mathbb{E}_{\boldsymbol{x},\mathrm{y}\sim\hat{p}_{\mathrm{data}}(\boldsymbol{x},y)}[L(f(\boldsymbol{x};\boldsymbol{\theta}),y)] = \frac{1}{m}\sum_{i=1}^{m} L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}),y^{(i)})$$

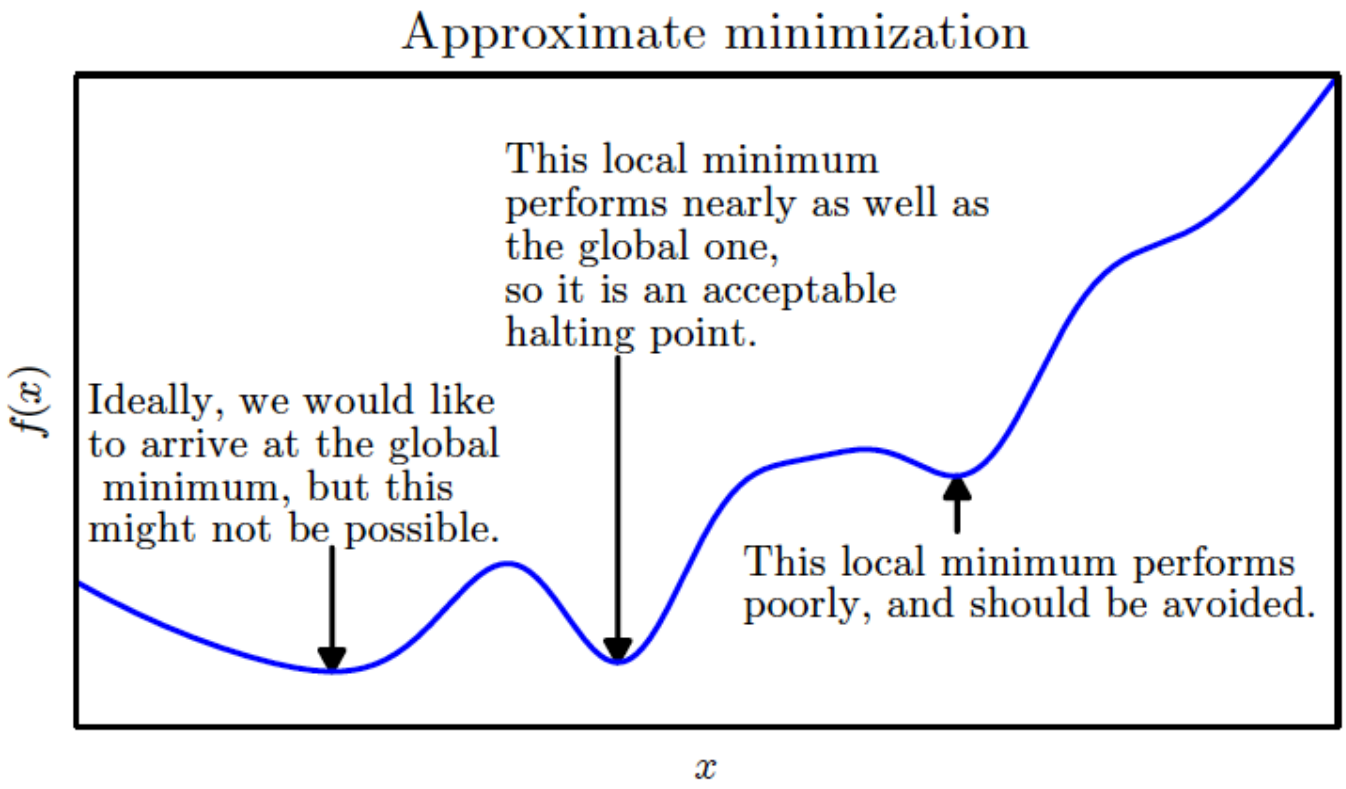
- Empirical risk minimization is prone to overfitting. Models with high capacity can simply memorize the training set.
- in the context of deep learning, we rarely use empirical risk minimization.

Loss Function

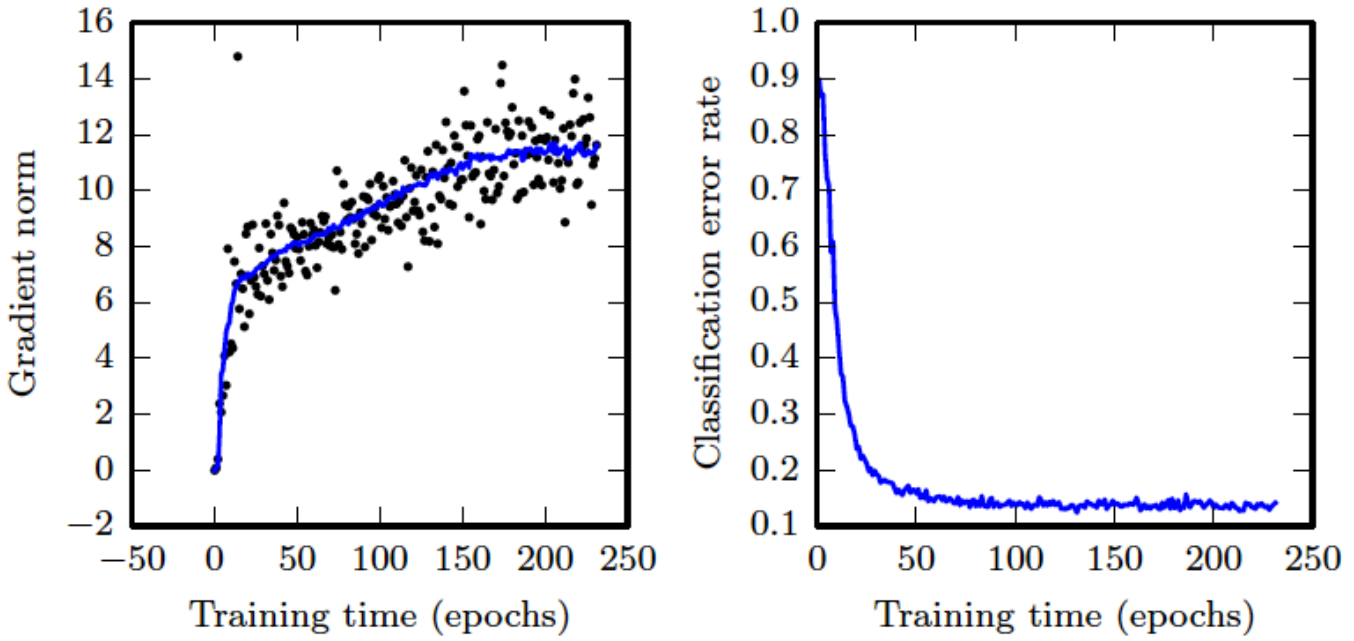
- The loss function we actually care about (say classification error) is notone that can be optimized efficiently.
  - Exactly minimizing expected 0-1 loss is typically intractable (exponential in the input dimension), even for a linear classifierIn such situations, oneoptimizes
- Optimise a surrogate loss function instead, which acts as a proxy but has advantages.
  - For example, the negative log-likelihood of the correct class is typically used as a surrogate for the 0-1 loss.
  - The negative log-likelihood allows the model to estimate the conditional probability of the classes, given the input.
- If network is well trained, then it can pick the classes that yield the least classification error in expectation.

Problem

- It is known that most complex Boolean functions require an exponential number of two-step logic gates for their representation (Wegener, 1987).
- The solution appears to be greater depth: according to Bengio (2014), the evidence strongly suggests that “functions that can be compactly represented with a depth-*k* architecture could require a very large number of elements in order to be represented by a shallower architecture”.


Approximate minimization
This local minimum performs nearly as well as the global one, so it is an acceptable halting point.
Ideally, we would like to arrive at the global minimum, but this might not be possible.
This local minimum performs poorly, and should be avoided.
$f(x)$
x

No Critical Point



Losses and regularization

- Logistic regression can be viewed as a simple neural network with no hidden units
- The underlying optimization criterion for predicting i=1,…,N labels y_i from features $\mathbf{x}_i$ with parameters θ consisting of a matrix of weights **W** and a vector of biases **b** can be viewed as

$$\sum_{i=1}^{N} -\log p(y_i \mid \mathbf{x}_i; \mathbf{W}, \mathbf{b}) + \lambda \sum_{j=1}^{M} w_j^2 = \sum_{i=1}^{N} L(f_i(\mathbf{x}_i; \theta), y_i) + \lambda R(\theta)$$

- where the first term, $L(f_i(\mathbf{x}_i;\theta), y_i)$, is the negative conditional log-likelihood or *loss*, and
- the second term, $\lambda R(\theta)$, is a weighted *regularizer* used to prevent overfitting

Common losses for neural networks

- The final output function of a neural network typically has the form $f_k(\mathbf{x})=f_k(a_k(\mathbf{x}))$, where $a_k(\mathbf{x})$ is just one of the elements of vector fuction $\mathbf{a}(\mathbf{x})=\mathbf{W}\mathbf{h}(\mathbf{x})+\mathbf{b}$
- Commonly used output loss functions, output activation functions, and the underlying distributions from which they derive are shown below

Loss Name, $L(f_i(\mathbf{x}_i;\theta),\mathbf{y}_i)=$	**Distribution Name**, $P(f_i(\mathbf{x}_i;\theta),\mathbf{y}_i)=$	**Output Activation Function**, $f_k(a_k(\mathbf{x}))=$
Squared error, $\sum_{k=1}^{K}(f_k(\mathbf{x})-y_k)^2$	Gaussian, $N(\mathbf{y};\mathbf{f}(\mathbf{x};\theta),\mathbf{I})$	$\frac{1}{(1+\exp(-a_k(\mathbf{x})))}$
Cross entropy, $-\sum_{k=1}^{K}[y_k\log f_k(\mathbf{x})+(1-y_k)\log(1-f_k(\mathbf{x}))]$	Bernoulli, $\mathrm{Bern}(\mathbf{y};\mathbf{f}(\mathbf{x};\theta))$	$\frac{1}{(1+\exp(-a_k(\mathbf{x})))}$
Softmax, $-\sum_{k=1}^{K}y_k\log f_k(\mathbf{x})$	Discrete or Categorical, $\mathrm{Cat}(\mathbf{y};\mathbf{f}(\mathbf{x};\theta))$	$\frac{\exp(a_k(\mathbf{x}))}{\sum_{j=1}^{K}\exp(a_j(\mathbf{x}))}$

Deep neural network architectures

- Compose computations performed by many layers
- Denoting the output of hidden layers by $\mathbf{h}^{(l)}(\mathbf{x})$, the computation for a network with *L* hidden layers is:

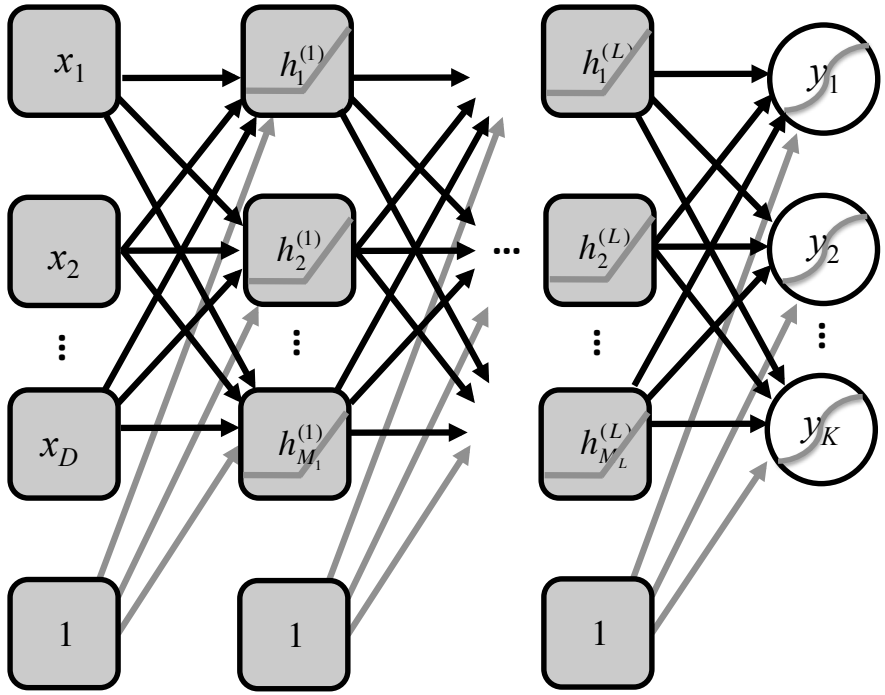
$$\mathbf{f}(\mathbf{x}) = \mathbf{f}\left[\mathbf{a}^{(L+1)}\left(\mathbf{h}^{(L)}\left(\mathbf{a}^{(L)}\left(\ldots\left(\mathbf{h}^{(2)}\left(\mathbf{a}^{(2)}\left(\mathbf{h}^{(1)}\left(\mathbf{a}^{(1)}(\mathbf{x})\right)\right)\right)\right)\right)\right)\right]$$

- Where *pre-activation functions* $\mathbf{a}^{(l)}(\mathbf{x})$ are typically linear, of the form $\mathbf{a}^{(l)}(\mathbf{x}) = \mathbf{W}^{(l)}\mathbf{x} + \mathbf{b}^{(l)}$ with matrix $\mathbf{W}^{(l)}$ and bias $\mathbf{b}^{(l)}$
- This formulation can be expressed using a single parameter matrix θ with the trick of defining $\hat{\mathbf{x}}$ as x with a 1 appended to the end of the vector; we then have

$$\mathbf{a}^{(l)}(\hat{\mathbf{x}}) = \boldsymbol{\theta}^{(l)}\hat{\mathbf{x}} \qquad , l = 1$$

$$\mathbf{a}^{(l)}(\hat{\mathbf{h}}^{(l-1)}) = \boldsymbol{\theta}^{(l)}\hat{\mathbf{h}}^{(l-1)} \qquad , l > 1$$

Deep feed-forward networks

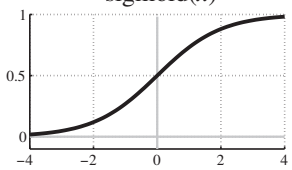
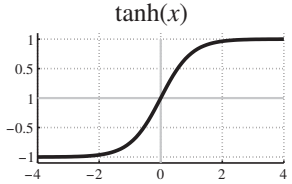
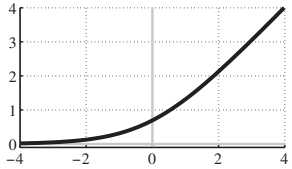
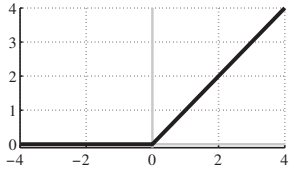


- Unlike Bayesian networks the hidden units here are *intermediate deterministic computations* not random variables, which is why they are not represented as circles
- However, the output variables y_k are drawn as circles because they can be formulated probabilistically

Activation functions

- Activation functions, $\mathbf{h}^{(l)}(\mathbf{x})$ generally operate on the pre-activation vectors in an *element-wise* fashion
- While sigmoid functions have been popular, the hyperbolic tangent function is sometimes preferred, partly because it has a steady state at 0
- More recently the *rectify*() function or rectified linear units (ReLUs) have been found to yield superior results in many different settings
  - Since ReLUs are 0 for negative argument values, some units in the model will yield activations that are 0, giving a sparseness property that is useful in many contexts
  - The gradient is particularly simple—either 0 or 1
  - This helps address the *exploding/diminishing gradient problem*

Activation functions

Name and Graph	Function	Derivative
sigmoid(x) 	$h(x) = \frac{1}{1 + \exp(-x)}$	$h'(x) = h(x)[1 - h(x)]$
tanh(x) 	$h(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}$	$h'(x) = 1 - h(x)^2$
softplus(x) 	$h(x) = \log(1 + \exp(x))$	$h'(x) = \frac{1}{1 + \exp(-x)}$
rectify(x) 	$h(x) = \max(0, x)$	$h'(x) = \begin{cases} 1 \text{ if } x \geq 0 \\ 0 \text{ if } x < 0 \end{cases}$

Back-propagation revisited in vector matrix form

Backpropagation in matrix vector form

- Backpropagation is based on the chain rule of calculus
- Consider the loss for a single-layer network with a softmax output (which corresponds exactly to the model for multinomial logistic regression)
- We use multinomial vectors **y**, with a single dimension $y_k = 1$ for the corresponding class label and whose other dimensions are 0
- Define $\mathbf{f} = [\ f_1(\mathbf{a}),\ \ldots,\ f_K(\mathbf{a})\ ]^T$ and $a_k(\mathbf{x};\theta_k) = \theta_k^T\mathbf{x}$, $\mathbf{a}(\mathbf{x};\theta) = [\ a_1(\mathbf{x};\theta_1),\ \ldots,\ a_K(\mathbf{x};\theta_K)\ ]^T$ where θ_k is a column vector containing the k^{th} row of the parameter matrix
- Consider the softmax loss for **f**(**a**(**x**))

Logistic regression and the chain rule

- Given loss $L = -\sum_{k=1}^{K} y_k \log f_k(\mathbf{x}), \quad f_k(\mathbf{x}) = \dfrac{\exp(a_k(\mathbf{x}))}{\sum_{c=1}^{K} \exp(a_c(\mathbf{x}))}.$
- Use the chain rule to obtain

$$\frac{\partial L}{\partial \boldsymbol{\theta}_k} = \frac{\partial \mathbf{a}}{\partial \boldsymbol{\theta}_k} \frac{\partial \mathbf{f}}{\partial \mathbf{a}} \frac{\partial L}{\partial \mathbf{f}} = \frac{\partial \mathbf{a}}{\partial \boldsymbol{\theta}_k} \frac{\partial L}{\partial \mathbf{a}}.$$

- Note the order of terms - in vector matrix form terms build from right to left

$$\frac{\partial L}{\partial a_j} = \frac{\partial}{\partial a_j}\left[-\sum_{k=1}^{K} y_k \left[a_k - \log\left[\sum_{c=1}^{K} \exp(a_c)\right]\right]\right]$$

$$= -\left[y_{k=j} - \frac{\exp(a_{k=j})}{\sum_{c=1}^{K} \exp(a_c)}\right] = -\left[y_j - p(y_j \mid \mathbf{x})\right] = -\left[y_j - f_j(\mathbf{x})\right],$$

Matrix vector form of gradient

- We can write $$\frac{\partial L}{\partial \mathbf{a}} = -\left[\mathbf{y} - \mathbf{f}(\mathbf{x})\right] \equiv -\Delta$$

  and since $$\frac{\partial a_j}{\partial \boldsymbol{\theta}_k} = \begin{cases} \dfrac{\partial}{\partial \boldsymbol{\theta}_k} \boldsymbol{\theta}_k^{\mathrm{T}} \mathbf{x} = \mathbf{x} & , j = k \\ 0 & , j \neq k \end{cases}$$

  we have $$\frac{\partial \mathbf{a}}{\partial \boldsymbol{\theta}_k} = \mathbf{H}_k = \begin{bmatrix} 0 & x_1 & 0 \\ \vdots & \vdots & \vdots \\ 0 & x_n & 0 \end{bmatrix}$$

- Notice that we avoid working with the partial derivative of the vector **a** with respect to the matrix θ, because it cannot be represented as a matrix — it is a multidimensional array of numbers (a tensor).

A compact expression for the gradient

- The gradient (as a column vector) for the vector in the *k*th row of the parameter matrix

$$\frac{\partial L}{\partial \boldsymbol{\theta}_k} = \frac{\partial \mathbf{a}}{\partial \boldsymbol{\theta}_k}\frac{\partial L}{\partial \mathbf{a}} = -\begin{bmatrix} 0 & x_1 & 0 \\ \vdots & \vdots & \vdots \\ 0 & x_n & 0 \end{bmatrix}\left[\mathbf{y} - \mathbf{f}(\mathbf{x})\right]$$

$$= -\mathbf{x}(y_k - f_k(x)).$$

- With a little rearrangement the gradient for the entire matrix of parameters can be written compactly:

$$\frac{\partial L}{\partial \boldsymbol{\theta}} = -\left[\mathbf{y} - \mathbf{f}(\mathbf{x})\right]\mathbf{x}^{\mathrm{T}} = -\Delta\mathbf{x}^{\mathrm{T}}.$$

Consider now a multilayer network

- Using the same activation function for all L hidden layers, and a softmax output layer
- The gradient of the k^{th} parameter vector of the $L{+}1^{\text{th}}$ matrix of parameters is

$$\frac{\partial L}{\partial \boldsymbol{\theta}_k^{(L+1)}} = \frac{\partial \mathbf{a}^{(L+1)}}{\partial \boldsymbol{\theta}_k^{(L+1)}} \frac{\partial L}{\partial \mathbf{a}^{(L+1)}}, \qquad \frac{\partial L}{\partial \mathbf{a}^{(L+1)}} = -\Delta^{(L+1)}$$

$$= -\frac{\partial \mathbf{a}^{(L+1)}}{\partial \boldsymbol{\theta}_k^{(L+1)}} \Delta^{(L+1)}$$

$$= -\mathbf{H}_k^L \Delta^{(L+1)} \quad \Rightarrow \quad \frac{\partial L}{\partial \boldsymbol{\theta}^{(L+1)}} = -\Delta^{(L+1)} \tilde{\mathbf{h}}_{(L)}^{\mathrm{T}}.$$

where $\mathbf{H}_k^L$ is a matrix containing the activations of the corresponding hidden layer, in column k

Backpropagating errors

- Consider the computation for the gradient of the k^{th} row of the L^{th} matrix of parameters
- Since the bias terms are constant, it is unnecessary to backprop through them, so

$$\begin{aligned}\frac{\partial L}{\partial\, \boldsymbol{\theta}_k^{(L)}} &= \frac{\partial\, \mathbf{a}^{(L)}}{\partial\, \boldsymbol{\theta}_k^{(L)}} \frac{\partial\, \mathbf{h}^{(L)}}{\partial\, \mathbf{a}^{(L)}} \frac{\partial\, \mathbf{a}^{(L+1)}}{\partial\, \mathbf{h}^{(L)}} \frac{\partial\, L}{\partial\, \mathbf{a}^{(L+1)}} \\ &= -\frac{\partial\, \mathbf{a}^{(L)}}{\partial\, \boldsymbol{\theta}_k^{(L)}} \frac{\partial\, \mathbf{h}^{(L)}}{\partial\, \mathbf{a}^{(L)}} \frac{\partial\, \mathbf{a}^{(L+1)}}{\partial\, \mathbf{h}^{(L)}} \Delta^{(L+1)}, \qquad \Delta^{(L)} \equiv \frac{\partial\, \mathbf{h}^{(L)}}{\partial\, \mathbf{a}^{(L)}} \frac{\partial\, \mathbf{a}^{(L+1)}}{\partial\, \mathbf{h}^{(L)}} \Delta^{(L+1)} \\ &= -\frac{\partial\, \mathbf{a}^{(L)}}{\partial\, \boldsymbol{\theta}_k^{(L)}} \Delta^{(L)}\end{aligned}$$

- Similarly, we can define $\Delta^{(l)}$ recursively in terms of $\Delta^{(l+1)}$

Backpropagating errors

- The backpropagated error can be written as simply

$$\Delta^{(l)} = \frac{\partial\, \mathbf{h}^{(l)}}{\partial\, \mathbf{a}^{(l)}} \frac{\partial\, \mathbf{a}^{(l+1)}}{\partial\, \mathbf{h}^{(l)}} \Delta^{(l+1)}, \qquad \frac{\partial\, \mathbf{h}^{(l)}}{\partial\, \mathbf{a}^{(l)}} = \mathbf{D}^{(l)}, \qquad \frac{\partial\, \mathbf{a}^{(l+1)}}{\partial\, \mathbf{h}^{(l)}} = \mathbf{W}^{\mathrm{T}(l+1)},$$

$$\Delta^{(l)} = \mathbf{D}^{(l)} \mathbf{W}^{\mathrm{T}(l+1)} \Delta^{(l+1)}$$

  where $\mathbf{D}^{(l)}$ contains the partial derivatives of the hidden-layer activation function with respect to the pre-activation input.

- $\mathbf{D}^{(l)}$ is generally diagonal, because activation functions usually operate on an element-wise basis
- $\mathbf{W}^{\mathrm{T}(l+1)}$ arises from the fact that $\mathbf{a}^{(l+1)}(\mathbf{h}^{(l)}) = \mathbf{W}^{(l+1)}\mathbf{h}^{(l)} + \mathbf{b}^{(l+1)}$

A general form for gradients

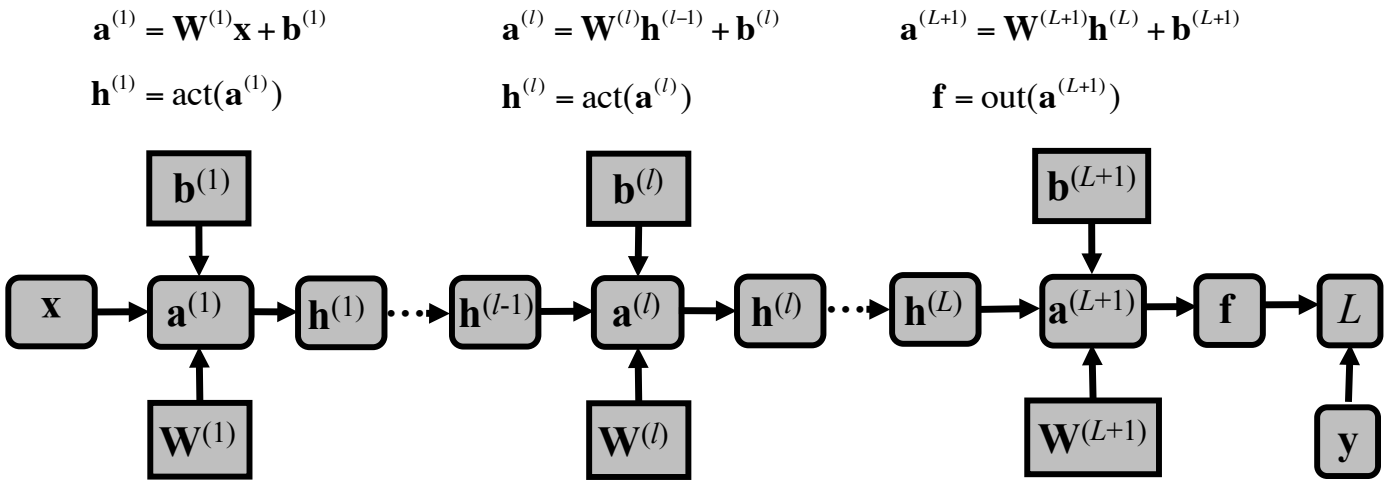
- The gradients for the k^{th} vector of parameters of the l^{th} network layer can therefore be computed using products of matrices of the following form

$$\frac{\partial L}{\partial \boldsymbol{\theta}_k^{(l)}} = -\mathbf{H}_k^{(l-1)}\mathbf{D}^{(l)}\mathbf{W}^{\mathrm{T}(l+1)}\cdots\mathbf{D}^{(L)}\mathbf{W}^{\mathrm{T}(L+1)}\Delta^{(L+1)}, \qquad \frac{\partial L}{\partial \boldsymbol{\theta}^{(l)}} = -\Delta^{(l)}\hat{\mathbf{h}}_{(l-1)}^{\mathrm{T}}$$

- When l=1, $\hat{\mathbf{h}}_{(0)} = \hat{\mathbf{x}}$, the input data with a 1 appended
- Note: since **D** is usually diagonal the corresponding matrix-vector multiply can be transformed into an element-wise product ∘ by extracting the diagonal for **d**

$$\Delta^{(l)} = \mathbf{D}^{(l)}(\mathbf{W}^{\mathrm{T}(l+1)}\Delta^{(l+1)}) = \mathbf{d}^{(l)} \circ (\mathbf{W}^{\mathrm{T}(l+1)}\Delta^{(l+1)})$$

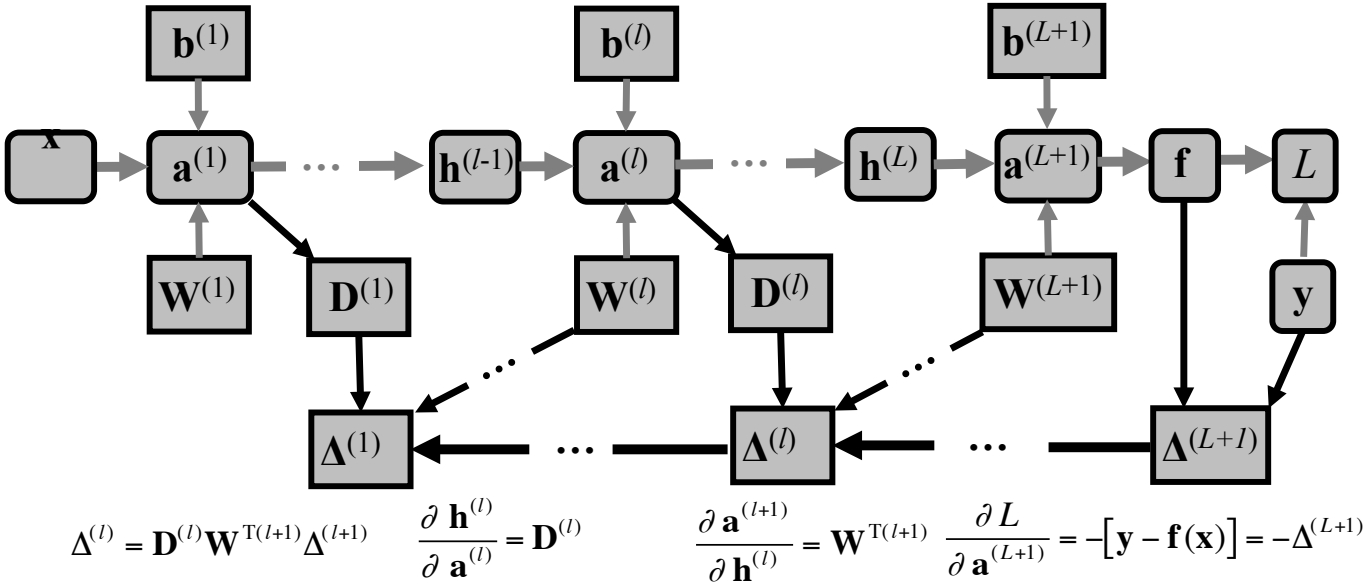
Visualizing backpropagation



- In the forward propagation phase we compute terms of the form above
- The figure above is a type of computation graph

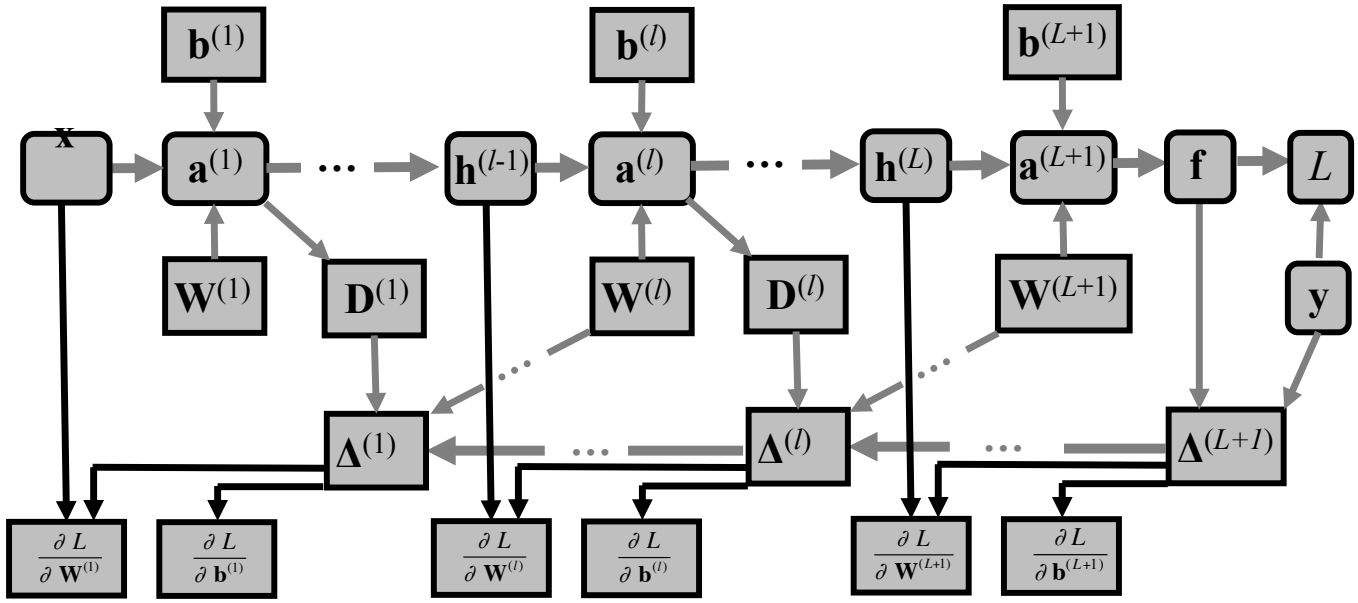
Visualizing backpropagation

- In the backward propagation phase we compute terms of the form below

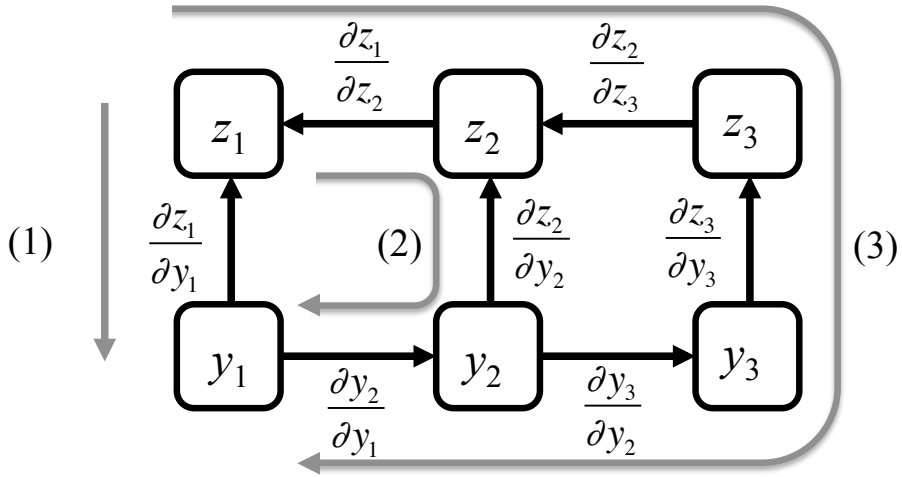


Visualizing backpropagation

- We update the parameters in our model using the simple computations below



Computation graphs



- For more complicated computations, computation graphs can help us keep track of how computations decompose, ex. $z_1 = z_1(y_1, z_2(y_2(y_1), z_3(y_3(y_2(y_1)))))$

$$\frac{\partial z_1}{\partial y_1} = \underbrace{\frac{\partial z_1}{\partial y_1}}_{(1)} + \underbrace{\frac{\partial z_1}{\partial z_2}\frac{\partial z_2}{\partial y_2}\frac{\partial y_2}{\partial y_1}}_{(2)} + \underbrace{\frac{\partial z_1}{\partial z_2}\frac{\partial z_2}{\partial z_3}\frac{\partial z_3}{\partial y_3}\frac{\partial y_3}{\partial y_2}\frac{\partial y_2}{\partial y_1}}_{(3)}$$

Checking an implementation of backpropagation and software tools

- An implementation of the backpropagation algorithm can be checked for correctness by comparing the analytic values of gradients with those computed numerically
- For example, one can add and subtract a small perturbation to each parameter and then compute the symmetric finite difference approximation to the derivative of the loss:

$$\frac{\partial L}{\partial \theta} \approx \frac{L(\theta+\varepsilon)-L(\theta-\varepsilon)}{2\varepsilon}$$

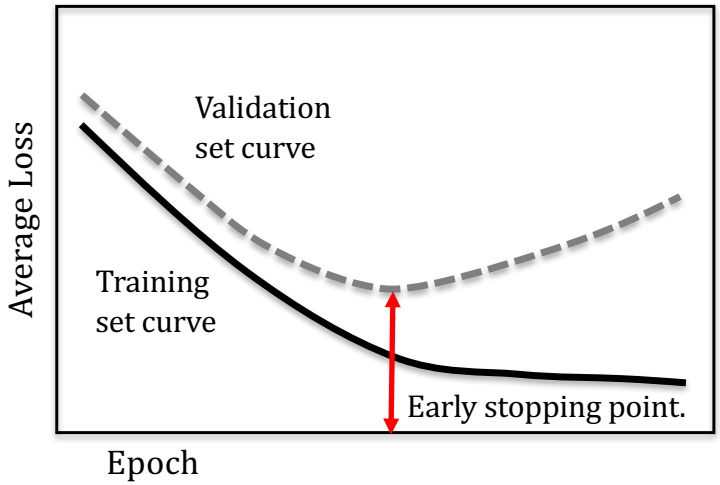
- Many software packages use computation graphs to allow complex networks to be more easily defined and optimized
- Examples include: Theano, TensorFlow, Keras and Torch

Training and evaluating deep networks

Early stopping

- Deep learning involves high capacity architectures, which are susceptible to overfitting even when data is plentiful,
- Early stopping is standard practice even when other methods to reduce overfitting are employed, ex. regularization and dropout
- The idea is to monitor learning curves that plot the average loss for the training and validation sets as a function of epoch
- The key is to find the point at which the validation set average loss begins to deteriorate

Early stopping



- In practice the curves above can be more noisy due to the use of stochastic gradient descent
- As such, it is common to keep the history of the validation set curve when looking for the minimum – even if it goes back up it might come back down

Validation sets and hyperparameters

- In deep learning hyperparameters are tuned by identifying what settings lead to best performance on the validation set, using early stopping
- Common hyperparameters include the strength of parameter regularization, but also model complexity in terms of the number of hidden units and layers and their connectivity, the form of activation functions, and parameters of the learning algorithm itself.
- Because of the many choices involved, performance monitoring on validation sets assumes an even more central role than it does with traditional machine learning methods.

Test sets

- *Should be set aside for a truly final evaluation*
- Repeated rounds of experiments using test set data give misleading (ex. optimistic) estimates of performance on fresh data
- For this reason, the research community has come to favor public challenges with hidden test-set labels, a development that has undoubtedly helped gauge progress in the field
- Controversy arises when participants submit multiple entries, and some favor a model where participants submit code to a competition server, so that the test data itself is hidden

Validation sets vs. cross-validation

- The use of a validation set is different from using *k*-fold cross-validation to evaluate a learning technique or to select hyperparameters.
- Cross-validation involves creating multiple training and testing partitions.
- Datasets for deep learning tend to be so massive that a single large test set adequately represents a model's performance, reducing the need for cross-validation
  - Since training often takes days or weeks, even using GPUs, cross-validation is often impractical anyway.
- If you do use cross validation you need to have an intern validation set *for each fold* to adjust hyperparameters or perform cross validation only using the training set

Validation set data and the ‘end game’

- To obtain the best possible results, one needs to tune hyperparameters, usually with a single validation set extracted from the training set.
- However, there is a dilemma: omitting the validation set from final training can reduce performance in the test.
- It is advantageous to train on the combined training and validation data, but this risks overfitting.
- One solution is to stop training after the same number of epochs that led to the best validation set performance; another is to monitor the average loss over the combined training set and stop when it reaches the level it was at when early stopping was performed using the validation set.
- One can use cross validation within the training set, treating each fold as a different validation set, then train the final model on the entire training data with the identified hyperparameters to perform the final test

Hyperparameter tuning

- A weighted combination of L_2 and L_1 regularization is often used to regularize weights
- Hyperparameters in deep learning are often tuned heuristically by hand, or using grid search
- An alternative is random search, where instead of placing a regular grid over hyperparameter space, probability distributions are specified from which samples are taken
- Another approach is to use machine learning and Bayesian techniques to infer the next hyperparameter configuration to try in a sequence of experimental runs
- Keep in mind even things like the learning rate schedule (discussed below) are forms of hyperparameters and you need to be careful not to tune them on the test set

Mini-batch based stochastic gradient descent (SGD)

- Stochastic gradient descent updates model parameters according to the gradient computed from one example
- The mini-batch variant uses a small subset of the data and bases updates to parameters on the average gradient over the examples in the batch
- This operates just like the regular procedure: initialize the parameters, enter a parameter update loop, and terminate by monitoring a validation set
- Normally these batches are randomly selected disjoint subsets of the training set, perhaps shuffled after each epoch, depending on the time required to do so

Mini-batch based SGD

- Each pass through a set of mini-batches that represent the complete training set is an *epoch*
- Using the empirical risk plus a regularization term as the objective function, updates are

$$\boldsymbol{\theta}^{\text{new}} \leftarrow \boldsymbol{\theta} - \eta_t \left[\frac{1}{B_k} \sum_{i \in \mathrm{I}} \left[\frac{\partial}{\partial \boldsymbol{\theta}} L(f(\mathbf{x}_i; \boldsymbol{\theta}), \mathbf{y}_i) \right] + \frac{B_k}{N} \lambda \frac{\partial}{\partial \boldsymbol{\theta}} R(\boldsymbol{\theta}) \right]$$

- η_t is the learning rate and may depend on the epoch t
- The batch is represented by a set of indices I=I(t,k) into the original data; the kth batch has B_k examples
- N is the size of the training set
- $L(f(\mathbf{x}_i;\theta),\mathbf{y}_i)$ is the loss for example $\mathbf{x}_i$, label $\mathbf{y}_i$, params θ
- R(θ) is the regularizer, with weight λ

Mini-batches

- Typically contain two to several hundred examples
  - For large models the choice may be constrained by resources
- Batch size often influences the stability and speed of learning; some sizes work particularly well for a given model and data set.
- Sometimes a search is performed over a set of potential batch sizes to find one that works well, before doing a lengthy optimization.
- The mix of class labels in the batches can influence the result
  - For unbalanced data there may be an advantage in pre-training the model using mini-batches in which the labels are balanced, then fine-tuning the upper layer or layers using the unbalanced label statistics.

Momentum

- As with regular gradient descent, 'momentum' can help the optimization escape plateaus in the loss
- Momentum is implemented by computing a moving average: $\Delta\theta = -\eta\nabla_{\theta}L(\theta) + \alpha\Delta\theta^{\text{old}}$
  - where the first term is the current gradient of the loss times a learning rate
  - the second term is the previous update weighted by $\alpha \in [0,1]$
- Since the mini- batch approach operates on a small subset of the data, this averaging can allow information from other recently seen mini-batches to contribute to the current parameter update
- A momentum value of 0.9 is often used as a starting point, but it is common to hand-tune it, the learning rate, and the schedule used to modify the learning rate during the training process

Learning rate schedules

- The learning rate is a critical choice when using mini-batch based stochastic gradient descent.
- Small values such as 0.001 often work well, but it is common to perform a logarithmically spaced search, say in the interval [10^{-8}, 1], followed by a finer grid or binary search.
- The learning rate may be adapted over epochs *t* to give a learning rate schedule, ex.

$$\eta_t = \eta_0 (1 + \varepsilon t)^{-1}$$

- A fixed learning rate is often used in the first few epochs, followed by a decreasing schedule
- Many other options, ex. divide the rate by 10 when the validation error rate ceases to improve

Mini-batch SGD pseudocode

$\theta = \theta_o$ // initialize parameters

$\Delta\theta = 0$

$t = 0$

while converged == FALSE

$\{I_1, \ldots, I_K\} = \text{shuffle}(X)$ // create K mini-batches

for $k = 1 \ldots K$

$$\mathbf{g} = \frac{1}{B_k} \sum_{i \in I_k} \left[\frac{\partial}{\partial \theta} L(f(\mathbf{x}_i; \theta), \mathbf{y}_i) \right] + \frac{B_k}{N} \lambda \frac{\partial}{\partial \theta} R(\theta)$$

$$\Delta\theta \leftarrow -\eta_t \mathbf{g} + \alpha \Delta\theta$$

$$\theta \leftarrow \theta + \Delta\theta$$

end

$t = t + 1$

end

Dropout

- A form of regularization that randomly deletes units and their connections during training
- Intention: reducing hidden unit co-adaptation & combat over-fitting
- Has been argued it corresponds to sampling from an exponential number of networks with shared parameters & missing connections
- One averages over models at test time by using original network without dropped-out connections, but with scaled-down weights
- If a unit is retained with probability p during training, its outgoing weights are rescaled or multiplied by a factor of p at test time
- By performing dropout a neural network with n units can be made to behave like an ensemble of 2^n smaller networks
- One way to implement it is with a binary mask vector $\mathbf{m}^{(l)}$ for each hidden layer l in the network: the dropped out version of $\mathbf{h}^{(l)}$ masks out units from the original version using element-wise multiplication, $\mathbf{h}_d^{(l)} = \mathbf{h}^{(l)} \odot \mathbf{m}^{(l)}$
- If the activation functions lead to diagonal gradient matrices, the backpropagation update is $\boldsymbol{\Delta}^{(l)} = \mathbf{d}^{(l)} \odot \mathbf{m}^{(l)} \odot (\mathbf{W}^{(l+1)} \boldsymbol{\Delta}^{(l+1)})$.

Batch normalization

- A way of accelerating training for which many studies have found to be important to obtain state-of-the-art results
- Each element of a layer is normalized to zero mean and unit variance based on its statistics within a mini-batch
  - This can change the network's representational power
- Each activation has learned scaling and shifting parameter
- Mini-batch based SGD is modified by calculating the mean μ_j and variance σ_j^2 over the batch for each hidden unit h_j in each layer, then normalize the units, scale them using the learned scaling parameter γ_j and shift them by the learned shifting parameter β_j such that

$$\hat{h}_j \leftarrow \gamma_j \frac{h_j - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}} + \beta_j .$$

- To update the γ_j and β_j one needs to backpropagate the gradient of the loss through these additional parameters

Parameter initialization

- Can be deceptively important!
- Bias terms are often initialized to 0 with no issues
- Weight matrices more problematic, ex.
  - If initialized to all 0s, can be shown that the tanh activation function will yield zero gradients
  - If the weights are all the same, hidden units will produce same gradients and behave the same as each other (wasting params)
- One solution: initialize all elements of weight matrix from uniform distribution over interval [–b, b]
- Different methods have been proposed for selecting the value of b, often motivated by the idea that units with more inputs should have smaller weights
- Weight matrices of rectified linear units have been successfully initialized using a zero-mean isotropic Gaussian distribution with standard deviation of 0.01

Unsupervised pre-training

- Idea: model the distribution of unlabeled data using a method that allows the parameters of the learned model to inform or be somehow transferred to the network
- Can be an effective way to both initialize and regularize a feedforward network
- Particularly useful when the volume of labeled data is small relative to the model's capacity.
- The use of activation functions such as rectified linear units (which improve gradient flow in deep networks), along with good parameter initialization techniques, can mitigate the need for sophisticated pre-training methods

Data augmentation

- Can be critical for best results
- As seen in MNIST table, augmenting even a large dataset with transformed data can increase performance
- A simple transformation is simply to jiggle the image
- If the object to be classified can be cropped out of a larger image, random bounding boxes can be placed around it, adding small translations in the vertical and horizontal directions
- Can also use rotations, scale changes, and shearing
- There is a hierarchy of rigid transformations that increase in complexity as parameters are added which can be used

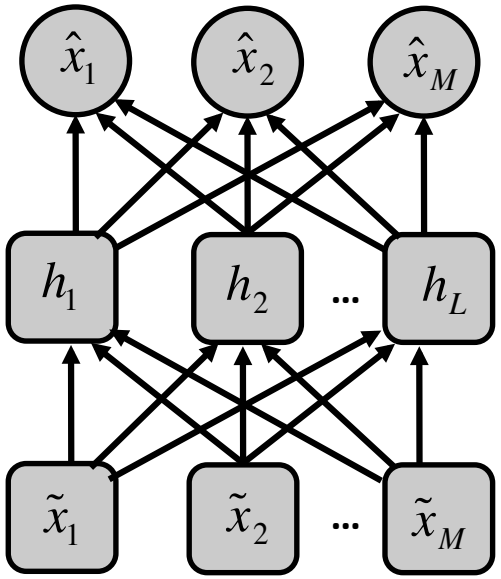
Autoencoders

Autoencoders

- Used for unsupervised learning
- It is a network that learns an efficient coding of its input.
- The objective is simply to reconstruct the input, but through the intermediary of a compressed or reduced-dimensional representation.
- If the output is formulated using probability, the objective function is to optimize $p(\mathbf{x} = \hat{\mathbf{x}} \mid \tilde{\mathbf{x}})$, that is, the probability that the model gives a random variable **x** the value $\hat{\mathbf{x}}$ given the observation $\tilde{\mathbf{x}}$, where $\hat{\mathbf{x}} = \tilde{\mathbf{x}}$.
- In other words, the model is trained to predict its own input—but it must map it through a representation created by the hidden units of a network.

A simple autoencoder

- Predicts its own input, ex. $p(\hat{\mathbf{x}} \mid \tilde{\mathbf{x}}) = p(\mathbf{x} = \hat{\mathbf{x}} \mid \tilde{\mathbf{x}}; \mathbf{f}(\tilde{\mathbf{x}}))$
- Going through an encoding, $\mathbf{e} = \mathbf{h} = \text{act}(\mathbf{a}^{(1)})$



where

$$\mathbf{f}(\tilde{\mathbf{x}}) = \mathbf{f}\left(\mathbf{d}\left(\mathbf{e}(\tilde{\mathbf{x}})\right)\right),$$

$$\mathbf{d} = \text{out}(\mathbf{a}^{(2)}),$$

$$\mathbf{a}^{(2)} = \mathbf{W}^{\mathrm{T}}\mathbf{h} + \mathbf{b}^{(2)},$$

$$\mathbf{h} = \text{act}(\mathbf{a}^{(1)}),$$

$$\mathbf{a}^{(1)} = \mathbf{W}\tilde{\mathbf{x}} + \mathbf{b}^{(1)}$$

Autoencoders

- Since the idea of an autoencoder is to compress the data into a lower-dimensional representation, the number *L* of hidden units used for encoding is less than the number *M* in the input and output layers
- Optimizing the autoencoder using the negative log probability over a data set as the objective function leads to the usual forms
- Like other neural networks it is typical to optimize autoencoders using backpropagation with mini-batch based SGD

Linear autoencoders and PCA

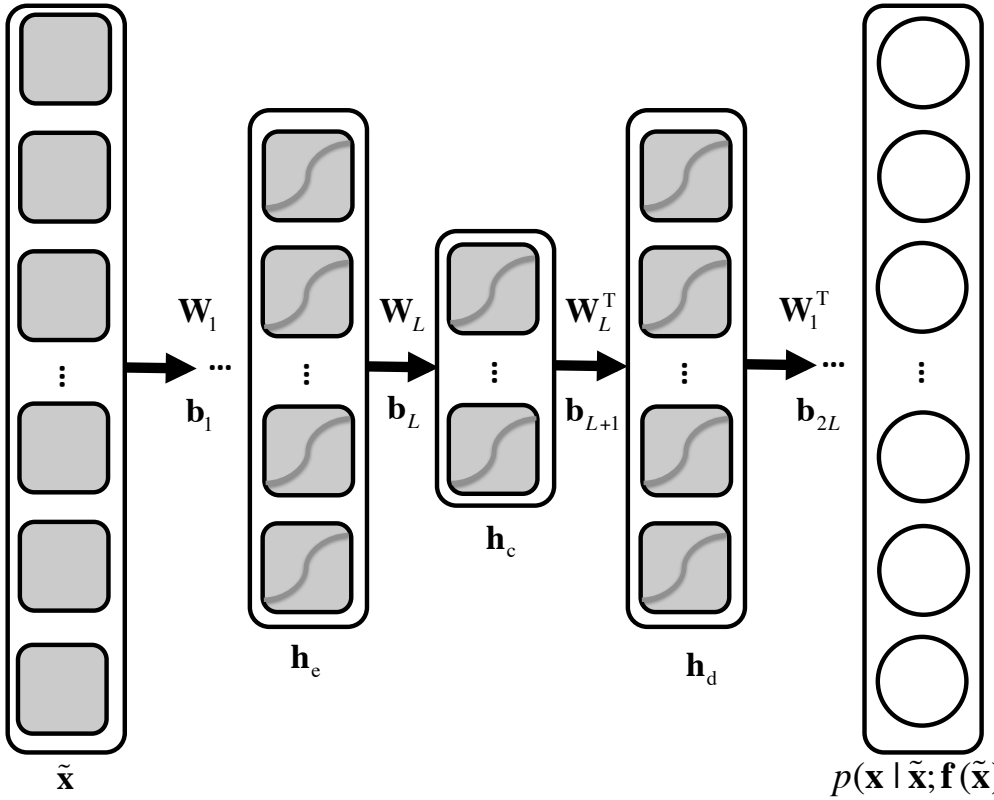
- Both the encoder activation function act() and the output activation function out() in the simple autoencoder model could be defined as the sigmoid function
- However, it can be shown that with no activation function, $\mathbf{h}^{(i)}=\mathbf{a}^{(i)}$, the resulting "linear autoencoder" will find the same subspace as PCA, (assuming a squared-error loss function and normalizing the data using mean centering)
  - Can be shown to be optimal in the sense that any model with a non-linear activation function would require a weight matrix with more parameters to achieve the same reconstruction error
- Even with non-linear activation functions such as a sigmoid, optimization finds solutions where the network operates in the linear regime, replicating the behavior of PCA
- This might seem discouraging; however, using a neural network with even one hidden layer to create much more flexible transformations, and
  - There is growing evidence deeper models can learn more useful representations

Deep autoencoders

- When building autoencoders from more flexible models, it is common to use a *bottleneck* in the network to produce an under-complete representation, providing a mechanism to obtain an encoding of lower dimension than the input.
- Deep autoencoders are able to learn low-dimensional representations with smaller reconstruction error than PCA using the same number of dimensions.
- Can be constructed by using L layers to create a hidden layer representation $\mathbf{h}_c^{(L)}$ of the data, and following this with a further L layers $\mathbf{h}_d^{(L+1)} \ldots \mathbf{h}_d^{(2L)}$ to decode the representation back into its original form
- The $j=1,\ldots,2L$ weight matrices for each of the $i=1,\ldots,L$ encoding and decoding layers are constrained by

$$\mathbf{W}_{L+i} = \mathbf{W}_{L+1-i}^{\mathrm{T}}$$

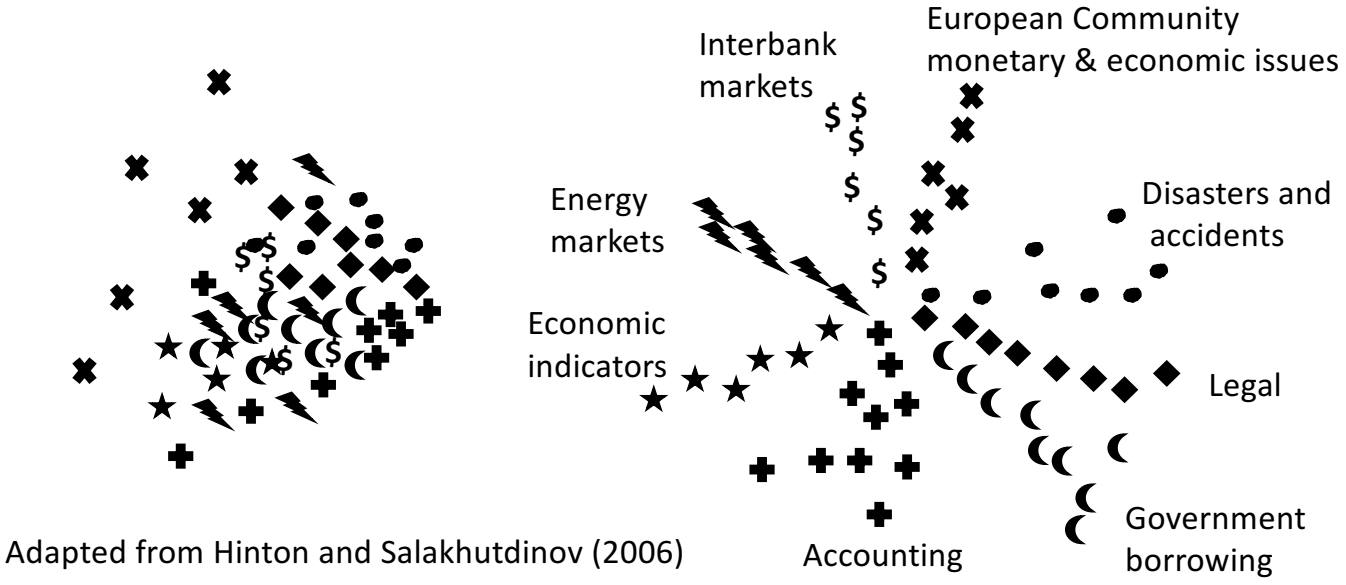
A deep autoencoder



$$\mathbf{f}(\mathbf{x}) = \mathbf{f}_{\mathbf{d}}(\mathbf{a}_{\mathbf{d}}^{(2L)}(\ldots\mathbf{h}_{\mathbf{d}}^{(L+1)}(\mathbf{a}_{\mathbf{d}}^{(L+1)}(\mathbf{h}_{\mathbf{c}}^{(L)}(\mathbf{a}_{\mathrm{e}}^{(L)}(\ldots\mathbf{h}_{\mathrm{e}}^{(1)}(\mathbf{a}_{\mathrm{e}}^{(1)}(\mathbf{x}))))))))).$$

Deep autoencoders

- A comparison of data projected into a 2D space with PCA (left) vs a deep autoencoder (right) for a text dataset
- The non-linear autoencoder can arrange the learned space in such that it better separates natural groupings of the data



Adapted from Hinton and Salakhutdinov (2006)

Training autoencoders

- Deep autoencoders are an effective framework for non-linear dimensionality reduction
- It can be difficult to optimize autoencoders; being careful about activation function choice and initialization can help
- Once such a network has been built, the top-most layer of the encoder, the code layer $\mathbf{h}_c$, can be input to a supervised classification procedure
- One can pre-train a discriminative neural net with an autoencoder
- One can also use a composite loss from the start, with a reconstructive (unsupervised) and a discriminative (supervised) criterion

$$L(\theta) = (1 - \lambda)L_{sup}(\theta) + \lambda L_{\text{unsup}}(\theta)$$

  where λ is a hyperparameter that balances the two objectives

- Another approach to training autoencoders is based on pre-training by stacking two-layered restricted Boltzmann machines RBMs

Stochastic methods

Boltzmann machines

- Are a type of Markov random field often used for unsupervised learning
- Unlike the units of a feedforward neural network, the units in Boltzmann machines correspond to random variables, such as are used in Bayesian networks
- Older variants of Boltzmann machines were defined using exclusively binary variables, but models with continuous and discrete variables are also possible
- They became popular prior to the impressive results of convolutional neural networks on the ImageNet challenge, but have since waned in popularity because they are more difficult to work with

Boltzmann machines

- To create a Boltzmann machine we partitioning variables into ones that are visible, using a *D*-dimensional binary vector **v**, and ones that are hidden, defined by a *K*-dimensional binary vector **h**
- A Boltzmann machine is a joint probability model of the form

$$p(\mathbf{v},\mathbf{h};\theta) = \frac{1}{Z(\theta)}\exp(-E(\mathbf{v},\mathbf{h};\theta)), \quad Z(\theta) = \sum_{\mathbf{v}}\sum_{\mathbf{h}}\exp(-E(\mathbf{v},\mathbf{h};\theta)),$$

$$E(\mathbf{v},\mathbf{h};\theta) = -\frac{1}{2}\mathbf{v}^T\mathbf{A}\mathbf{v} - \frac{1}{2}\mathbf{h}^T\mathbf{B}\mathbf{h} - \mathbf{v}^T\mathbf{W}\mathbf{h} - \mathbf{a}^{\mathrm{T}}\mathbf{v} - \mathbf{b}^{\mathrm{T}}\mathbf{h},$$

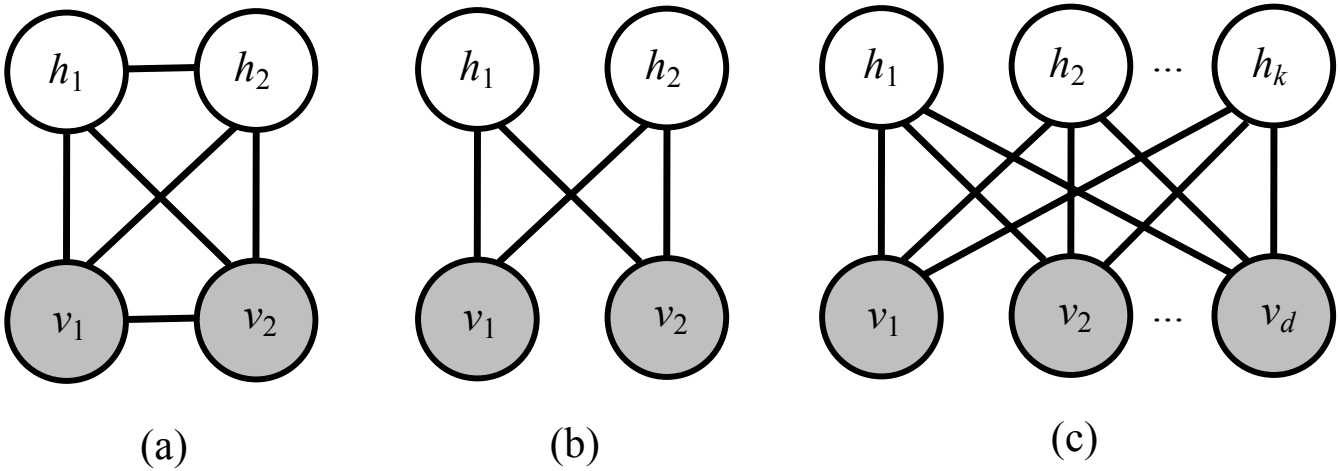
- where E(**v**,**h**;θ) is the energy function
- Z(θ) normalizes *E* so that it defines a valid joint probability
- matrices **A**, **B** and **W** encode the visible-to-visible, hidden-to-hidden and the visible-to-hidden variable interactions respectively
- vectors **a** and **b** encode the biases associated with each variable
- matrices **A** and **B** are symmetric, and their diagonal elements are 0

Boltzmann machines

(a) Boltzman Machines are binary Markov random field with pairwise connections between all variables

(b) Restricted Boltzmann machines (RBMs) do not have connections between the variables in a layer

(c) RBMs can be extended to many variables as shown



Key feature of Boltzmann machines

- A key feature of Boltzmann machines (and binary Markov random fields in general) is that the conditional distribution of one variable given the others is a sigmoid function whose argument is a weighted linear combination of the states of the other variables

$$p(h_j = 1 \mid \mathbf{v}, \mathbf{h}_{\neg j}; \theta) = \text{sigmoid}\left(\sum_{i=1}^{D} W_{ij} v_i + \sum_{k=1}^{K} B_{jk} h_k + b_j \right),$$

$$p(v_i = 1 \mid \mathbf{h}, \mathbf{v}_{\neg i}; \theta) = \text{sigmoid}\left(\sum_{j=1}^{K} W_{ij} h_j + \sum_{d=1}^{D} A_{id} v_d + c_i \right),$$

where the notation $\neg i$ indicates all elements with subscript other than *i*.

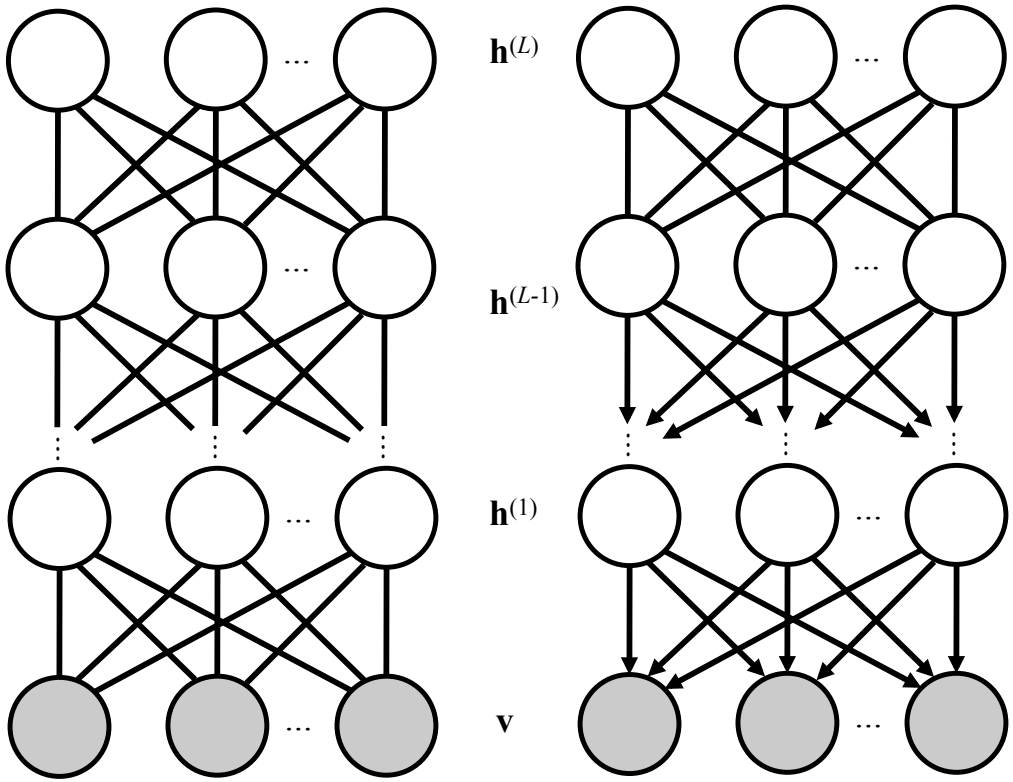
Contrastive divergence

- Running a Gibbs sampler for a Boltzmann machine often requires many iterations,
- A technique called “contrastive divergence” is a popular alternative that initializes the sampler to the observed data instead of randomly and performs a limited number of Gibbs updates.
- In an RBM a sample can be generated from the sigmoid distributions for all the hidden variables given the observed; then samples can be generated for the observed variables given the hidden variable sample
- This single step often works well in practice, although the process of alternating the sampling of hidden and visible units can be continued for multiple steps.

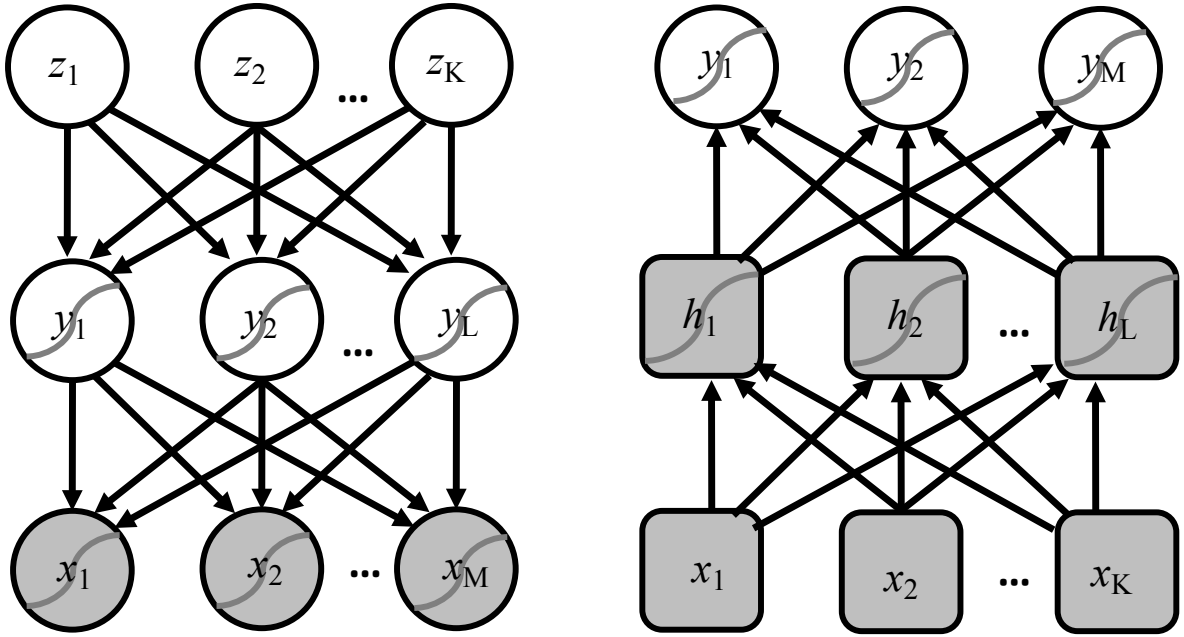
Deep RBMs and deep belief networks

- Deep Boltzmann machines involve coupling layers of random variables using restricted Boltzmann machine connectivity
- While any deep Bayesian network is technically a deep belief network, the term “deep belief network” has become strongly associated with a particular type of deep architecture that can be constructed by training restricted Boltzmann machines incrementally.
- The procedure is based on converting the lower part of a growing model into a Bayesian belief network, adding an RBM for the upper part of the model, then continuing the training, conversion and stacking process.

A deep RBM vs a deep belief network



A sigmoidal belief network vs a neural network



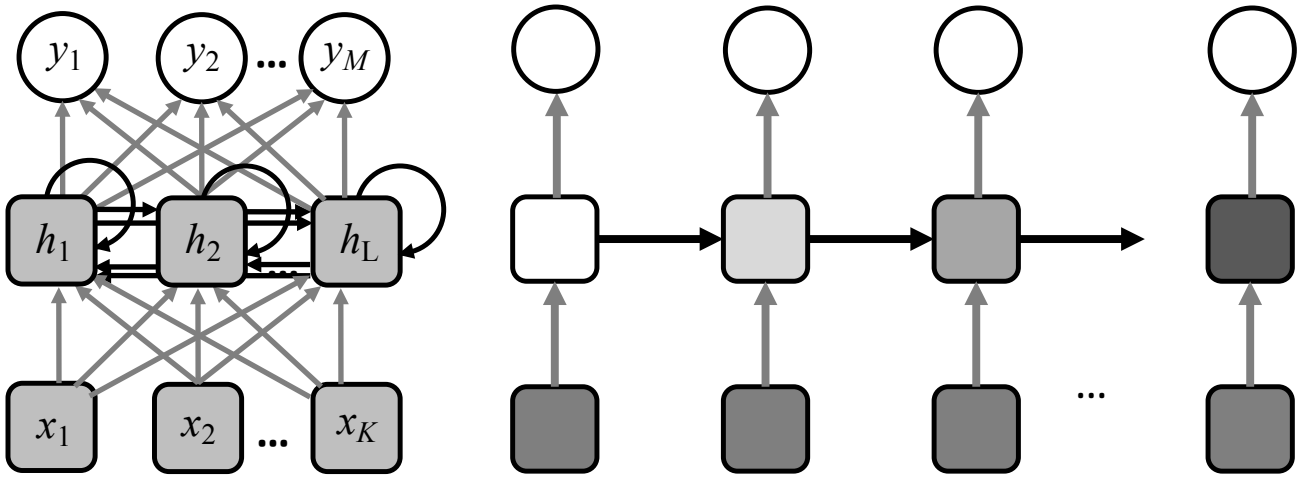
Recurrent neural networks

Recurrent neural networks

- Recurrent neural networks are networks with connections that form directed cycles.
- As a result, they have an internal state, which makes them prime candidates for tackling learning problems involving sequences of data—such as handwriting recognition, speech recognition, and machine translation.
- A feedforward network can be transformed into a recurrent network by adding connections from all hidden units h_i to h_j.
- Each hidden unit has connections to both itself and other hidden units.
- Imagine unfolding a recurrent network over time by following the sequence of steps that perform the underlying computation.
- Like a hidden Markov model, a recurrent network can be unwrapped and implemented using the same weights and biases at each step to link units over time.

Recurrent neural networks (RNNs)

- An RNN can be unwrapped and implemented using the same weights and biases at each step to link units over time as shown below
- The resulting unwrapped RNN is similar to a hidden Markov model, but keep in mind that the hidden units in RNNs are not stochastic



Recurrent neural networks (RNNs)

- Recurrent neural networks apply linear matrix operations to the current observation and the hidden units from the previous time step, and the resulting linear terms serve as arguments of activation functions act():

$$\mathbf{h}_t = \text{act}(\mathbf{W}_h \mathbf{x}_t + \mathbf{U}_h \mathbf{h}_{t-1} + \mathbf{b}_h)$$

$$\mathbf{o}_t = \text{act}(\mathbf{W}_o \mathbf{h}_t + \mathbf{b}_o)$$

- The same matrix $\mathbf{U}_h$ is used at each time step
- The hidden units in the previous step $\mathbf{h}_{t-1}$ influence the computation of $\mathbf{h}_t$ where the current observation contributes to a $\mathbf{W}_h\mathbf{x}$ term that is combined with $\mathbf{U}_h\mathbf{h}_{t-1}$ and bias $\mathbf{b}_h$ terms
- Both $\mathbf{W}_h$ and $\mathbf{b}_h$ are typically replicated over time
- The output layer is modeled by a classical neural network activation function applied to a linear transformation of the hidden units, the operation is replicated at each step.

The loss, exploding and vanishing gradients

- The loss for a particular sequence in the training data can be computed either at each time step or just once, at the end of the sequence.
- In either case, predictions will be made after many processing steps and this brings us to an important problem.
- The gradient for feedforward networks decomposes the gradient of parameters at layer l into a term that involves the product of matrix multiplications of the form $\mathbf{D}^{(l)}\mathbf{W}^{\mathsf{T}(l+1)}$ (see the analysis for feedforward networks above)
- A recurrent network uses the same matrix at each time step, and over many steps the gradient can very easily either diminish to zero or explode to infinity—just as the magnitude of any number other than one taken to a large power either approaches zero or increases indefinitely

Dealing with exploding gradients

- The use of L_1 or L_2 regularization can mitigate the problem of exploding gradients by encouraging weights to be small.
- Another strategy is to simply detect if the norm of the gradient exceeds some threshold, and if so, scale it down.
- This is sometimes called gradient (norm) clipping where for a gradient vector $\mathbf{g}$ and threshold T,

$$\text{if } \|\mathbf{g}\| \geq T \text{ then } \mathbf{g} \leftarrow \frac{T}{\|\mathbf{g}\|}\mathbf{g}$$

where T is a hyperparameter, which can be set to the average norm over several previous updates where clipping was not used.

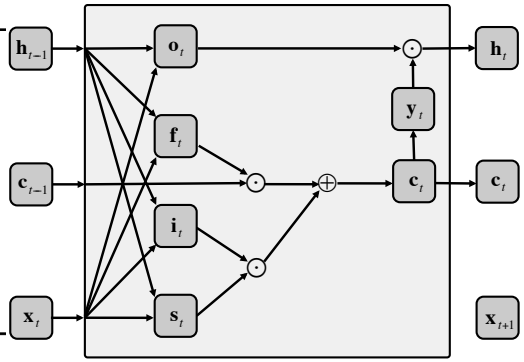
LSTMs and vanishing gradients

- The so-called “long short term memory” (LSTM) RNN architecture was specifically created to address the vanishing gradient problem.
- Uses a combination of hidden units, element wise products and sums between units to implement gates that control “memory cells”.
- Memory cells are designed to retain information without modification for long periods of time.
- They have their own input and output gates, which are controlled by learnable weights that are a function of the current observation and the hidden units at the previous time step.
- As a result, *backpropagated error terms from gradient computations can be stored and propagated backwards without degradation*.
- The original LSTM formulation consisted of *input gates* and *output gates*, but *forget gates* and “peephole weights” were added later.
- Below we present the most popular variant of LSTM RNNs which does not include peephole weights, but which does use forget gates.
- The architecture is complex, but has produced state-of-the-art results on a wide variety of problems.

LSTM architecture

- At each time step there are three types of gates: input $\mathbf{i}_t$, forget $\mathbf{f}_t$, and output $\mathbf{o}_t$.
- Each are a function of both the underlying input $\mathbf{x}_t$ at time *t* as well as the hidden units at time *t*-1, $\mathbf{h}_{t-1}$
- Each gate multiplies $\mathbf{x}_t$ by its own gate specific **W** matrix, by its own **U** matrix, and adds its own bias vector **b**.
- This is usually followed by the application of a sigmoidal element wise non-linearity.

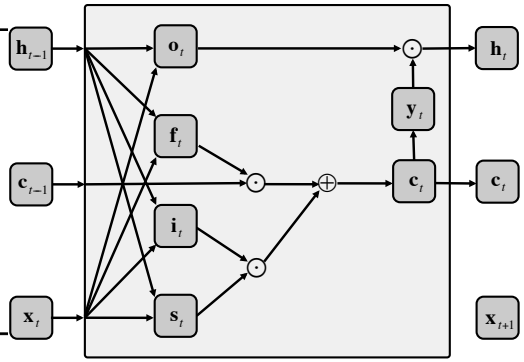
LSTM unit output	$\mathbf{h}_t = \mathbf{o}_t \circ \mathbf{y}_t$
Output gate units	$\mathbf{o}_t = \text{sigmoid}(\mathbf{W}_o\mathbf{x}_t + \mathbf{U}_o\mathbf{h}_{t-1} + \mathbf{b}_o)$
Transformed memory cell contents	$\mathbf{y}_t = \tanh(\mathbf{c}_t)$
Gated update to memory cell units	$\mathbf{c}_t = \mathbf{f}_t \circ \mathbf{c}_{t-1} + \mathbf{i}_t \circ \mathbf{s}_t$
Forget gate units	$\mathbf{f}_t = \text{sigmoid}(\mathbf{W}_f\mathbf{x}_t + \mathbf{U}_f\mathbf{h}_{t-1} + \mathbf{b}_f)$
Input gate units	$\mathbf{i}_t = \text{sigmoid}(\mathbf{W}_i\mathbf{x}_t + \mathbf{U}_i\mathbf{h}_{t-1} + \mathbf{b}_i)$
Potential input to memory cell	$\mathbf{s}_t = \tanh(\mathbf{W}_c\mathbf{x}_t + \mathbf{U}_c\mathbf{h}_{t-1} + \mathbf{b}_c)$



LSTM architecture

- At each time step *t*, input gates $\mathbf{i}_t$ are used to determine when a potential input given by $\mathbf{s}_t$ is important enough to be placed into the memory unit or cell, $\mathbf{c}_t$
- Forget gates $\mathbf{f}_t$ allow memory unit content to be erased
- Output gates $\mathbf{o}_t$ determine whether $\mathbf{y}_t$, the content of the memory units transformed by activation functions, should be placed in the hidden units $\mathbf{h}_t$
- Typical gate activation functions and their dependencies are shown below

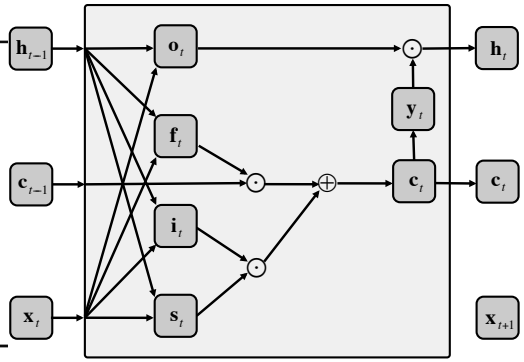
LSTM unit output	$\mathbf{h}_t = \mathbf{o}_t \circ \mathbf{y}_t$
Output gate units	$\mathbf{o}_t = \text{sigmoid}(\mathbf{W}_o\mathbf{x}_t + \mathbf{U}_o\mathbf{h}_{t-1} + \mathbf{b}_o)$
Transformed memory cell contents	$\mathbf{y}_t = \tanh(\mathbf{c}_t)$
Gated update to memory cell units	$\mathbf{c}_t = \mathbf{f}_t \circ \mathbf{c}_{t-1} + \mathbf{i}_t \circ \mathbf{s}_t$
Forget gate units	$\mathbf{f}_t = \text{sigmoid}(\mathbf{W}_f\mathbf{x}_t + \mathbf{U}_f\mathbf{h}_{t-1} + \mathbf{b}_f)$
Input gate units	$\mathbf{i}_t = \text{sigmoid}(\mathbf{W}_i\mathbf{x}_t + \mathbf{U}_i\mathbf{h}_{t-1} + \mathbf{b}_i)$
Potential input to memory cell	$\mathbf{s}_t = \tanh(\mathbf{W}_c\mathbf{x}_t + \mathbf{U}_c\mathbf{h}_{t-1} + \mathbf{b}_c)$



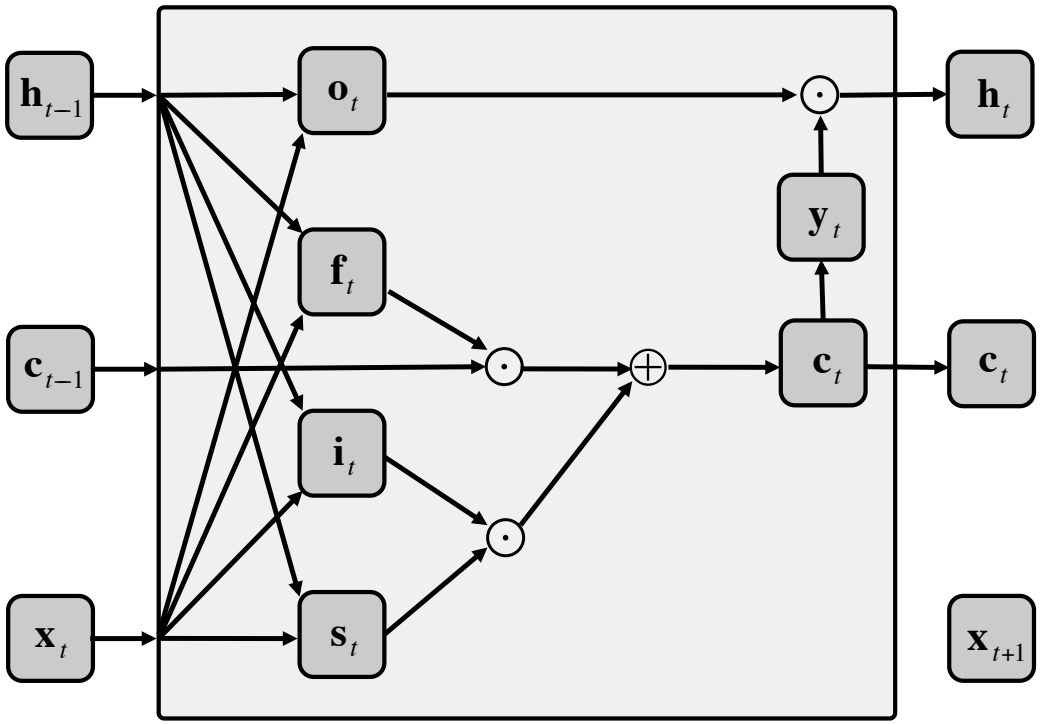
LSTM achitecture

- The final gating is implemented as an elementwise product between the output gate and the transformed memory contents, $\mathbf{h}_t = \mathbf{o}_t \circ \mathbf{y}_t$
- Memory units are typically transformed by the tanh function prior to the gated output, such that $\mathbf{y}_t = \tanh(\mathbf{c}_t)$
- Memory units or cells are updated by $\mathbf{c}_t = \mathbf{f}_t \circ \mathbf{c}_{t-1} + \mathbf{i}_t \circ \mathbf{s}_t$ an elementwise product between the forget gates and the previous contents of the memory units, plus the elementwise product of the input gates and the new potential inputs .

LSTM unit output	$\mathbf{h}_t = \mathbf{o}_t \circ \mathbf{y}_t$
Output gate units	$\mathbf{o}_t = \text{sigmoid}(\mathbf{W}_o \mathbf{x}_t + \mathbf{U}_o \mathbf{h}_{t-1} + \mathbf{b}_o)$
Transformed memory cell contents	$\mathbf{y}_t = \tanh(\mathbf{c}_t)$
Gated update to memory cell units	$\mathbf{c}_t = \mathbf{f}_t \circ \mathbf{c}_{t-1} + \mathbf{i}_t \circ \mathbf{s}_t$
Forget gate units	$\mathbf{f}_t = \text{sigmoid}(\mathbf{W}_f \mathbf{x}_t + \mathbf{U}_f \mathbf{h}_{t-1} + \mathbf{b}_f)$
Input gate units	$\mathbf{i}_t = \text{sigmoid}(\mathbf{W}_i \mathbf{x}_t + \mathbf{U}_i \mathbf{h}_{t-1} + \mathbf{b}_i)$
Potential input to memory cell	$\mathbf{s}_t = \tanh(\mathbf{W}_c \mathbf{x}_t + \mathbf{U}_c \mathbf{h}_{t-1} + \mathbf{b}_c)$



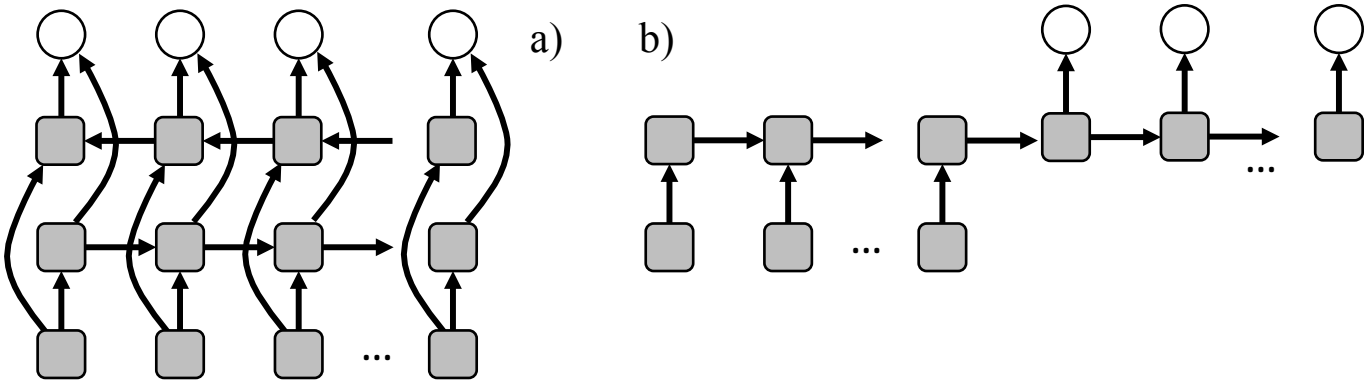
LSTM achitecture



Other RNN architectures

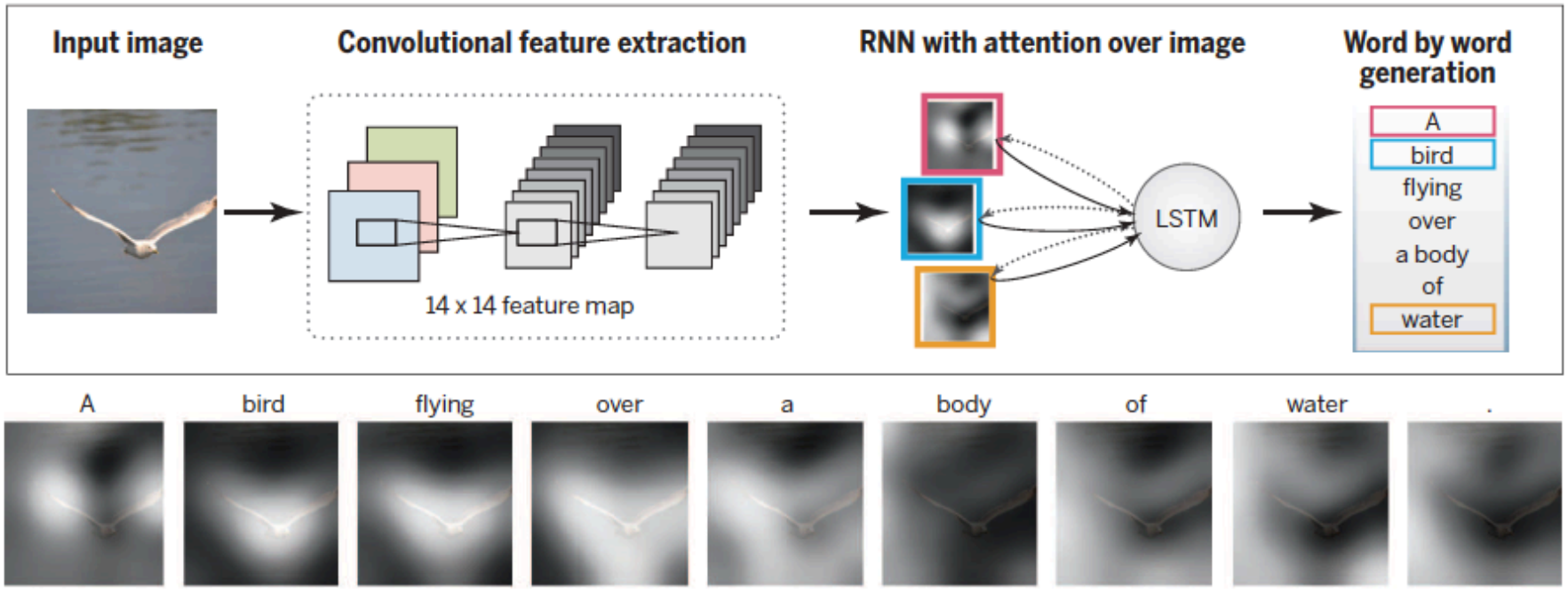
a) Recurrent networks can be made bidirectional, propagating information in both directions
   - They have been used for a wide variety of applications, including protein secondary structure prediction and handwriting recognition

b) An “encoder-decoder” network creates a fixed-length vector representation for variable-length inputs, the encoding can be used to generate a variable-length sequence as the output
   - Particularly useful for machine translation



Looking At Some Recent Interesting Developments

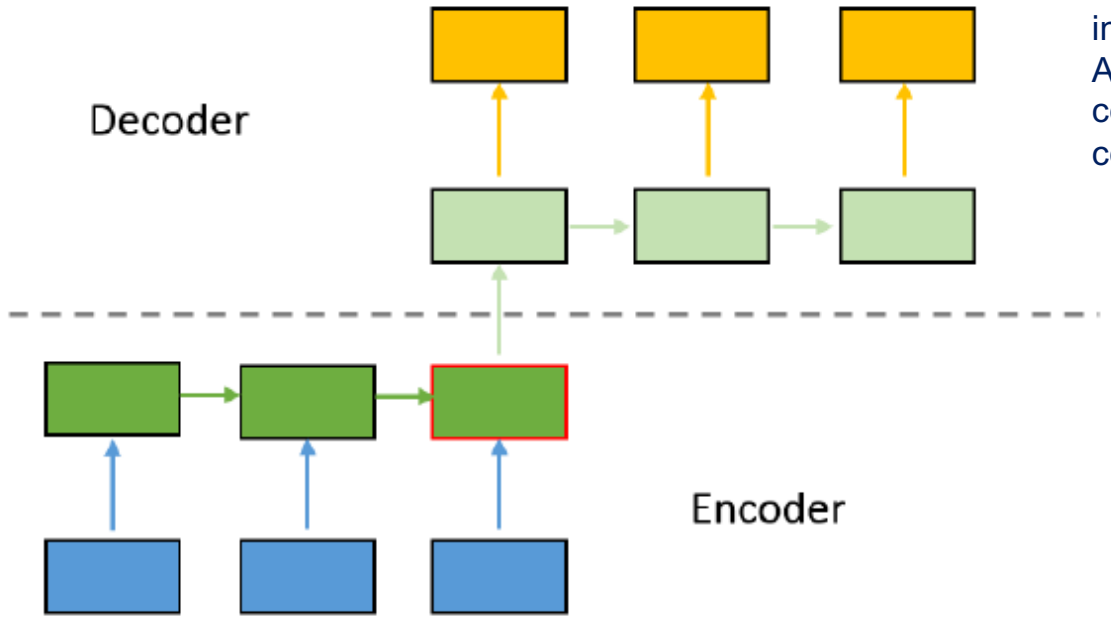
- Attention
- Deep Network with memory
- Understanding Brain
- Designing Deep Network
- Deep Learning with non-stationary data
- Efficient Deep Learning



Automatic generation of text captions for images with deep networks. A convolutional neural network is trained to interpret images, and its output is then used by a recurrent neural network trained to generate a text caption (top). The sequence at the bottom shows the word-by-word focus of the network on different parts of input image while it generates the caption word-by-word.

Attention

- Main aspects of attention
  - Decide which part of the input needs to be focused on
  - Allocate the limited processing resources to the important part
- In human visual processing, although human's eye has the ability to receive a large visual field, usually only a small part is fixated on.
- Some RNN models are designed for sequence to sequence problems with the attention mechanism.
- RNN can operate over sequences of input vectors, but also generate sequences of output vectors.



The encoder and the decoder. The rectangle surrounded by the red box is the intermediate code. The blue rectangles represent the input items, the dark green rectangles are the hidden states of the encoder, the shallow green rectangles represent the hidden states of the decoder, and the yellow rectangles are the output items.

The model does not try to figure out the corresponding relations between the items in the input and output sequences,
All items in the input sequence are compressed into a single intermediate code.

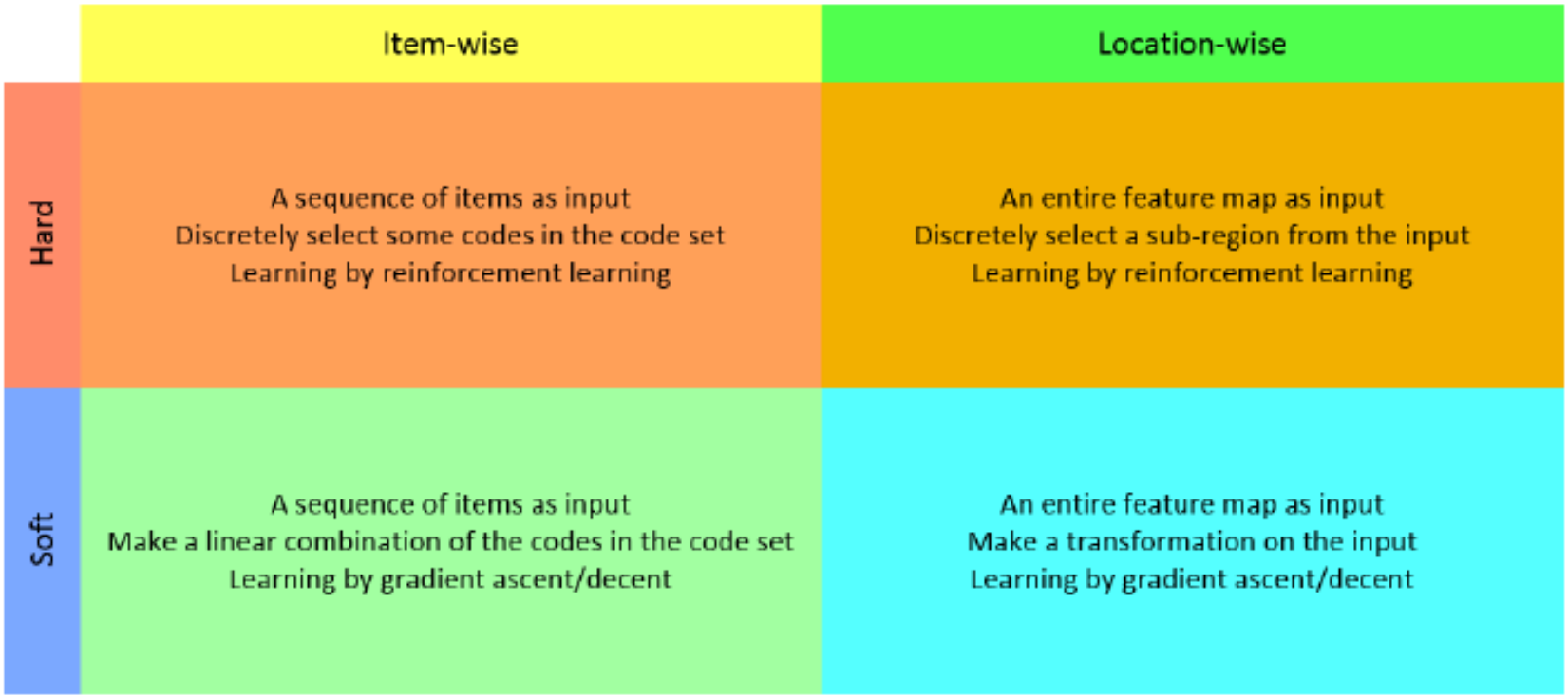
The length of the intermediate code is fixed, which prevents the model giving different weights to different items in an input sequence explicitly, so all items in the input sequence has the same importance no matter which output item is attempted to be predicted.

This inspires the researchers to add the attention mechanism into the common RNN model.

Attention RNN

- With the attention mechanism, the RNN model is able to assign different weights to different parts of items in the input sequence.
  - The inherent corresponding relations between items in input and output sequences can be captured and exploited explicitly.
- The attention module is an additional neural network which is connected to the original RNN model
- Four types of attention mechanisms
  - item-wise soft attention
  - item-wise hard attention
  - location-wise hard attention
  - location-wise soft attention

Attention Mechanism



Advantages

- The attention based RNN model is able to learn to assign weights to different parts of the input instead of treating all input items equally
  - Exploit the inherent relations between the items the in input and output sequence
  - Can boost the performance of some tasks
- The hard attention model does not need to process all items in the input sequence, instead, sometimes only processes the interested ones, so it is very useful for some tasks with only partially observable environments, like game playing, which usually cannot be handled by the common RNN model.
- Not limited in computer vision area, the attention based RNN model is also suitable for all kinds of sequence related problems. For example:
  - Applications in natural language processing (NLP)
    - Machine translation
    - Machine comprehension
    - sentence summarization
    - word representation
  - Bio-informatics
  - Speech recognition
  - Game play
  - Robotics.

Soft Attention

- The item-wise soft attention requires the input sequence X contains some explicit items $\{x_1;x_2;:::;x_T\}$.
- Instead of extracting only one code c from the input X, the encoder of the item-wise soft attention RNN model extracts a set of codes C from X

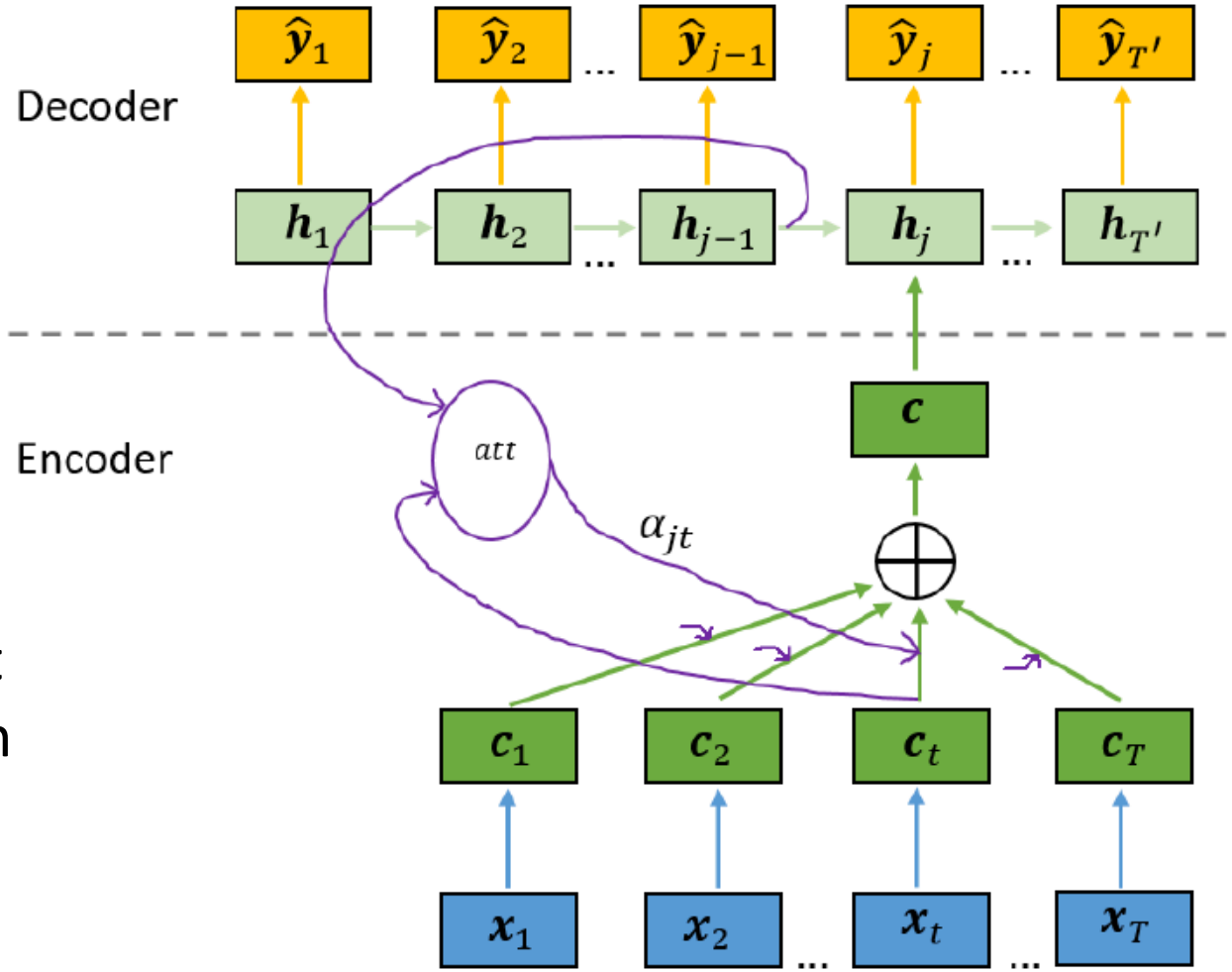
$$C = \{\boldsymbol{c}_1, \boldsymbol{c}_2, \ldots, \boldsymbol{c}_{T''}\} \qquad \boldsymbol{c}_t = \phi_{W_{enc}}(\boldsymbol{x}_t) \text{ for } t \in (1, 2, \ldots, T)$$

- During the decoding, the intermediate code c fed into the decoder is calculated by the attention module, which accepts all individual codes C, and the decoder's previous hidden state h_{j-1} as input. At the decoding step j, the intermediate code is:

$$\boldsymbol{c} = \boldsymbol{c}^j = \phi_{W_{att}}(C, \boldsymbol{h}_{j-1})$$

- Item wise weight is calculated as $e_{jt} = f_{att}(\boldsymbol{c}_t, \boldsymbol{h}_{j-1})$ $\alpha_{jt} = \dfrac{\exp(e_{jt})}{\sum_{t=1}^{T} \exp(e_{jt})}$
- Attention function should be differentiable

Item wise soft attention model



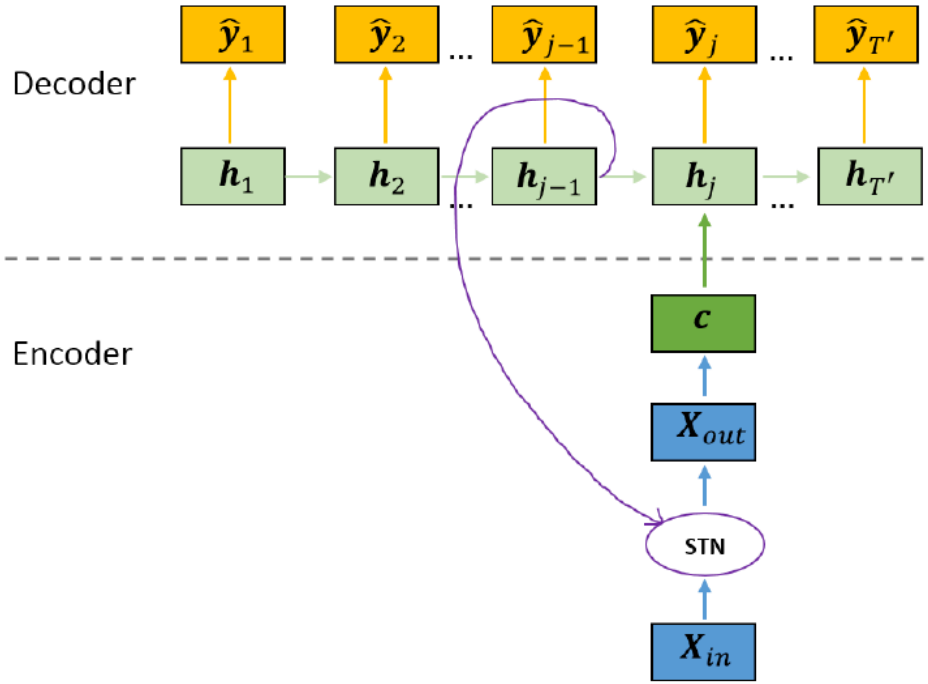
Hard Attention Model

The item-wise hard attention is very similar to the item-wise soft attention. It still needs to calculate the weights for each code as shown from Equation (9) to Equation (13). As mentioned above, the α_{jt} can be interpreted as the probability that how the code $\boldsymbol{c}_t$ is relevant to the output $\boldsymbol{y}_j$, so instead of a linear combination of all codes in C, the item-wise hard attention stochastically picks one code based on their probabilities. In detail, an indicator l_j is generated from a categorical distribution at decoding step j to indicate which code should be picked:

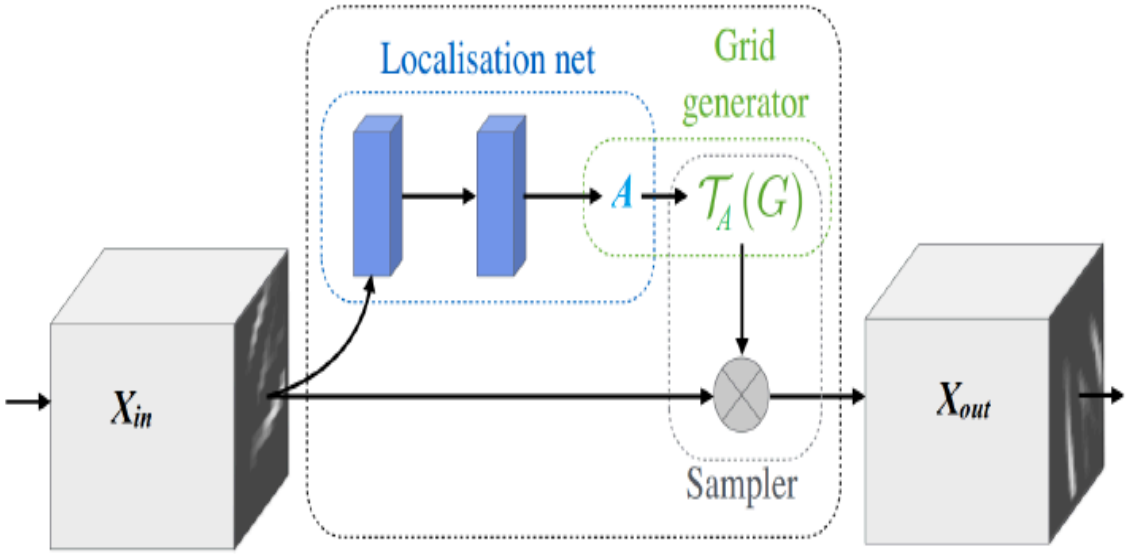
$$l_j \sim \mathcal{C}\left(T, \{\alpha_{jt}\}_{t=1}^{T}\right)$$

Location Wise Soft Attention

- STN (Spatial transformation network)
  - Accepts a feature map X as input, and at each decoding step, a transformed version of the input feature map is generated to calculate the intermediate code.



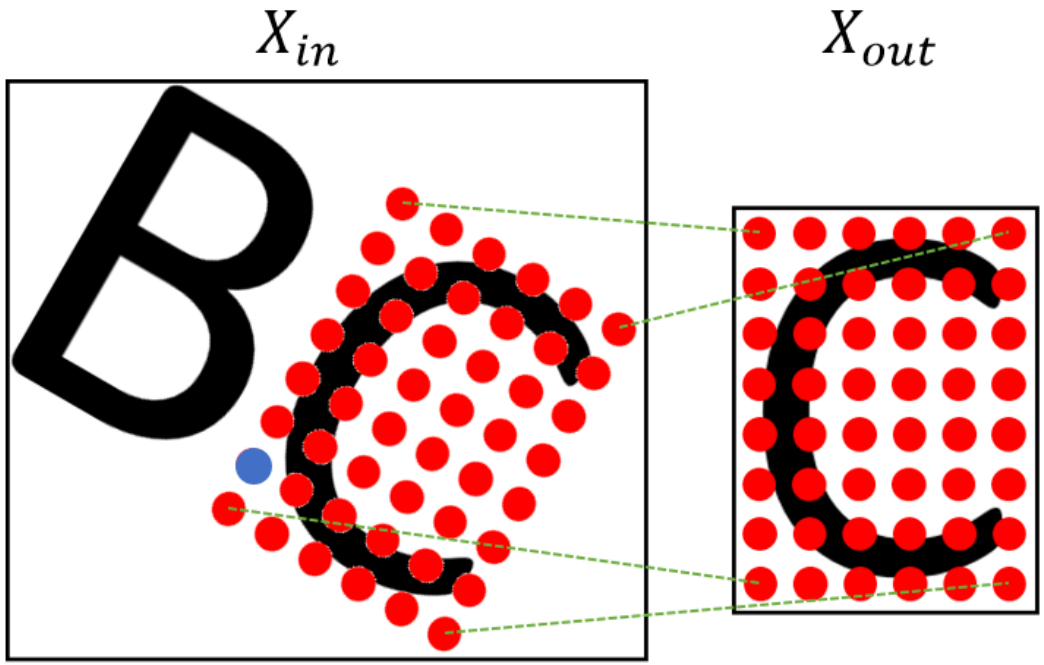
STN



At each decoding time j, the localization network $\phi_{W_{loc}}$ takes the input X_{in}, the previous hidden state of the decoder $\boldsymbol{h}_{j-1}$ as input, and generates the transformation parameter A_j as output:

$$A_j = \phi_{W_{loc}}(X_{in}, \boldsymbol{h}_{j-1}) \tag{25}^{2}$$

Rectified Text



Learning Attention Model

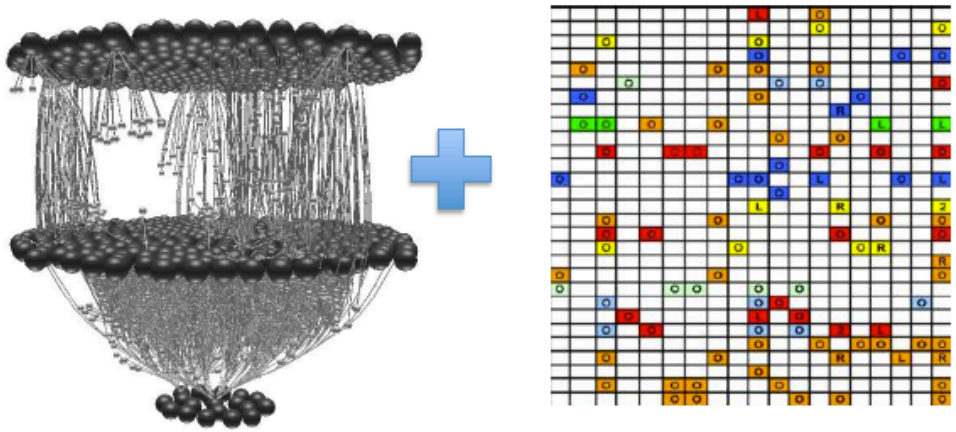
- the difference between the soft and hard attention mechanisms in optimization is that the soft attention module is differentiable with respect to its parameters so the standard gradient ascent/decent can be used for optimization.
- However, for the hard attention, the model is non-differentiable and the techniques from the reinforcement learning are applied for optimization.

The location-wise soft attention module (STN) is also differentiable with respect to its parameters if the location network $\phi_{W_{loc}}$, the transformation τ and the sampling kernel k are carefully selected to ensure their gradients with respect to their inputs can be defined.

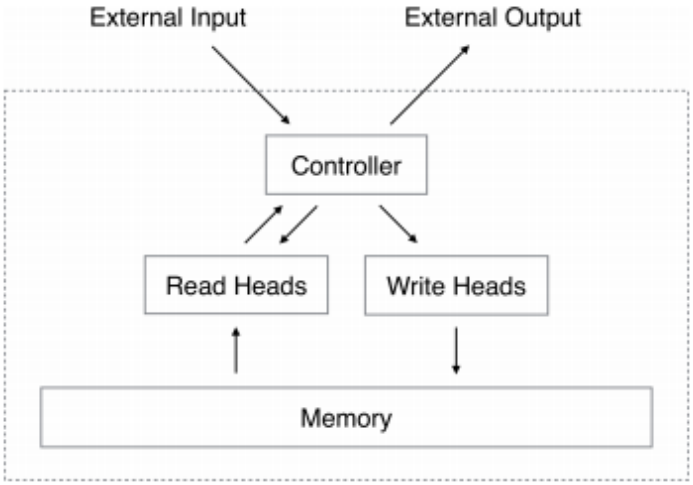
Open Questions

- How to use multiple attention modules in parallel?
- Can we generalize RNN and make it have a similar structure as the modern computer architecture, a multiple cache system where the attention module serves as a "scheduler" or an addressing module?

Neural Turing Machine



Extend the Capabilities of neural networks by coupling them to external memory



1. Reading
   - M_t is NxM matrix of memory at time t
   - w_t

$$\sum_i w_t(i) = 1, \qquad 0 \le w_t(i) \le 1, \forall i.$$

$$\mathbf{r}_t \longleftarrow \sum_i w_t(i)\mathbf{M}_t(i),$$

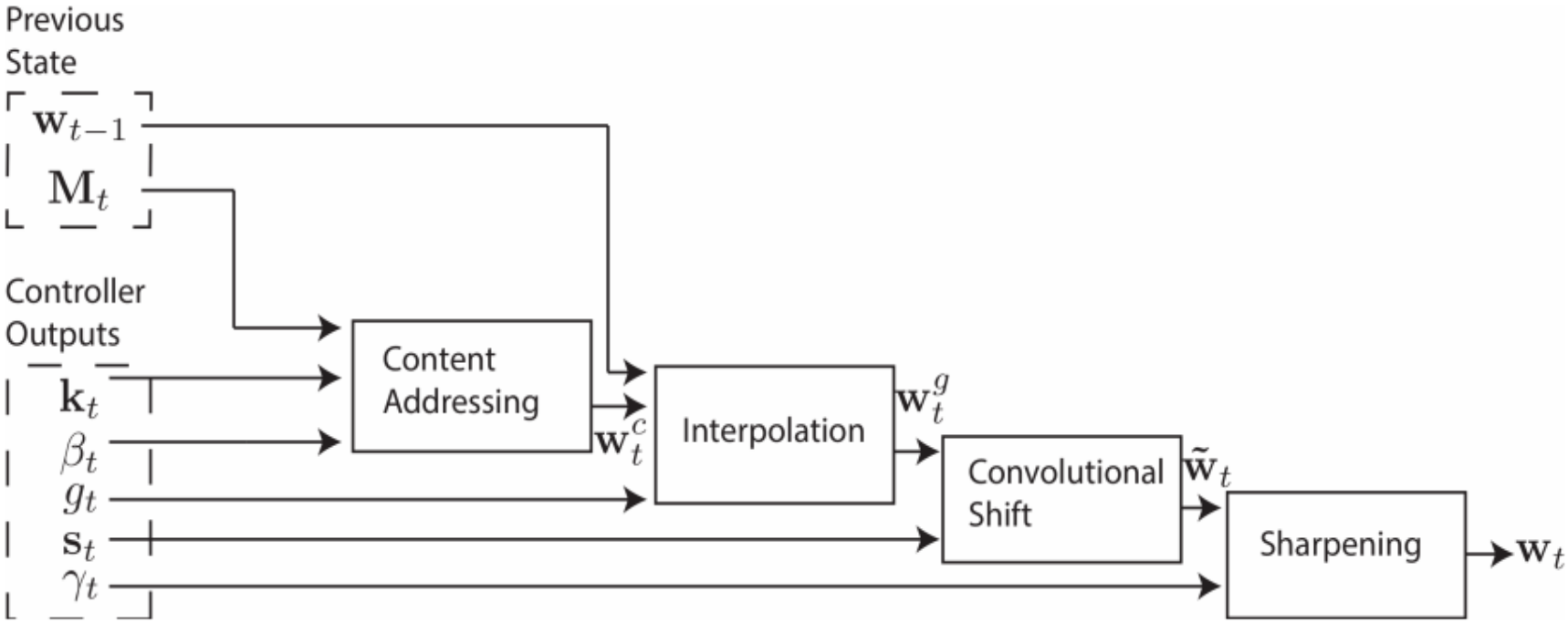
2. Writing involves both erasing and adding

$$\tilde{\mathbf{M}}_t(i) \longleftarrow \mathbf{M}_{t-1}(i)\left[\mathbf{1} - w_t(i)\mathbf{e}_t\right],$$

$$\mathbf{M}_t(i) \longleftarrow \tilde{\mathbf{M}}_t(i) + w_t(i)\,\mathbf{a}_t.$$

3. Addressing

Addressing


Previous
State
$\mathbf{w}_{t-1}$
$\mathbf{M}_t$
Controller
Outputs
$\mathbf{k}_t$
β_t
g_t
$\mathbf{s}_t$
γ_t
Content
Addressing
$\mathbf{w}_t^c$
Interpolation
$\mathbf{w}_t^g$
Convolutional
Shift
$\tilde{\mathbf{w}}_t$
Sharpening
$\mathbf{w}_t$

Addressing

– 1. Focusing by Content

- Each head produces key vector $\mathbf{k_t}$ of length M
- Generated a content based weight $\mathbf{w_t^c}$ based on similarity measure, using 'key strength' β_t

$$w_t^c(i) \longleftarrow \frac{\exp\left(\beta_t K\left[\mathbf{k}_t, \mathbf{M}_t(i)\right]\right)}{\sum_j \exp\left(\beta_t K\left[\mathbf{k}_t, \mathbf{M}_t(j)\right]\right)}.$$

$$K\left[\mathbf{u}, \mathbf{v}\right] = \frac{\mathbf{u} \cdot \mathbf{v}}{||\mathbf{u}|| \cdot ||\mathbf{v}||}.$$

Addressing

– 2. Interpolation

- Each head emits a scalar interpolation gate g_t

$$\mathbf{w}_t^g \longleftarrow g_t \mathbf{w}_t^c + (1 - g_t)\mathbf{w}_{t-1}.$$

Addressing

– 3. Convolutional shift

- Each head emits a distribution over allowable integer shifts s_t

$$\tilde{w}_t(i) \longleftarrow \sum_{j=0}^{N-1} w_t^g(j)\, s_t(i-j)$$

Addressing

– 4. Sharpening

- Each head emits a scalar sharpening parameter γ_t

$$w_t(i) \longleftarrow \frac{\tilde{w}_t(i)^{\gamma_t}}{\sum_j \tilde{w}_t(j)^{\gamma_t}}$$

Addressing: Putting All Together

- This can operate in three complementary modes
  - A weighting can be chosen by the content system without any modification by the location system
  - A weighting produced by the content addressing system can be chosen and then shifted
  - A weighting from the previous time step can be rotated without any input from the content-based addressing system

Controller Architecture

- Feed Forward vs Recurrent
- The LSTM version of RNN has own internal memory complementary to M
- Hidden LSTM layers are ‘like’ registers in processor
- Allows for mix of information across multiple time-steps
- Feed Forward has better transparency

Application: Associative Recall

- Tests NTM's ability to associate data references
- Training input is list of items, followed by a query item
- Output is subsequent item in list
- Each item is a three sequence 6-bit binary vector
- Each 'episode' has between two and six items

Associative Recall

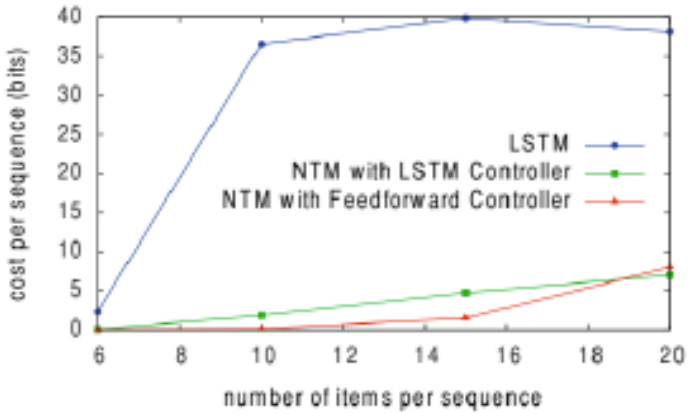


Figure 11: Generalisation Performance on Associative Recall for Longer Item Sequences. The NTM with either a feedforward or LSTM controller generalises to much longer sequences of items than the LSTM alone. In particular, the NTM with a feedforward controller is nearly perfect for item sequences of twice the length of sequences in its training set.

Open Issues

- Alternatives
- Theory for Computational Power

Understanding Brain

RNN for Modeling Ganglion

Sensory Processing

- Developing accurate predictive models of sensory neurons is vital to understanding sensory processing and brain computations.
- Multilayer recurrent neural networks that share features across neurons outperform generalized linear models (GLMs) in predicting the spiking responses of parasol ganglion cells in the primate retina to natural images.
- The networks achieve good predictive performance given surprisingly small amounts of experimental training data.
- Additionally, a GLM-RNN hybrid model with separate spatial and temporal processing components provides insights into the aspects of retinal processing better captured by the recurrent neural networks.

In a **general linear model**

$$y_i = \beta_0 + \beta_1 x_{1i} + ... + \beta_p x_{pi} + \epsilon_i$$

the **response** $y_i, i = 1, \ldots, n$ is modelled by a linear function of **explanatory** variables $x_j, j = 1, \ldots, p$ plus an error term.

Here **general** refers to the dependence on potentially more than one explanatory variable, v.s. the **simple linear model**:

$$y_i = \beta_0 + \beta_1 x_i + \epsilon_i$$

The model is *linear in the parameters*, e.g.

$$y_i = \beta_0 + \beta_1 x_1 + \beta_2 x_1^2 + \epsilon_i$$
$$y_i = \beta_0 + \gamma_1 \delta_1 x_1 + \exp(\beta_2) x_2 + \epsilon_i$$

Generalized Linear Models (GLMs)

A **generalized linear model** is made up of a **linear predictor**

$$\eta_i = \beta_0 + \beta_1 x_{1i} + ... + \beta_p x_{pi}$$

and two functions

- a **link** function that describes how the mean, $E(Y_i) = \mu_i$, depends on the linear predictor

$$g(\mu_i) = \eta_i$$

- a **variance** function that describes how the variance, $\text{var}(Y_i)$ depends on the mean

$$\text{var}(Y_i) = \phi V(\mu)$$

where the **dispersion parameter** ϕ is a constant

Modelling Poisson Data

Suppose

$$Y_i \sim \text{Poisson}(\lambda_i)$$

Then

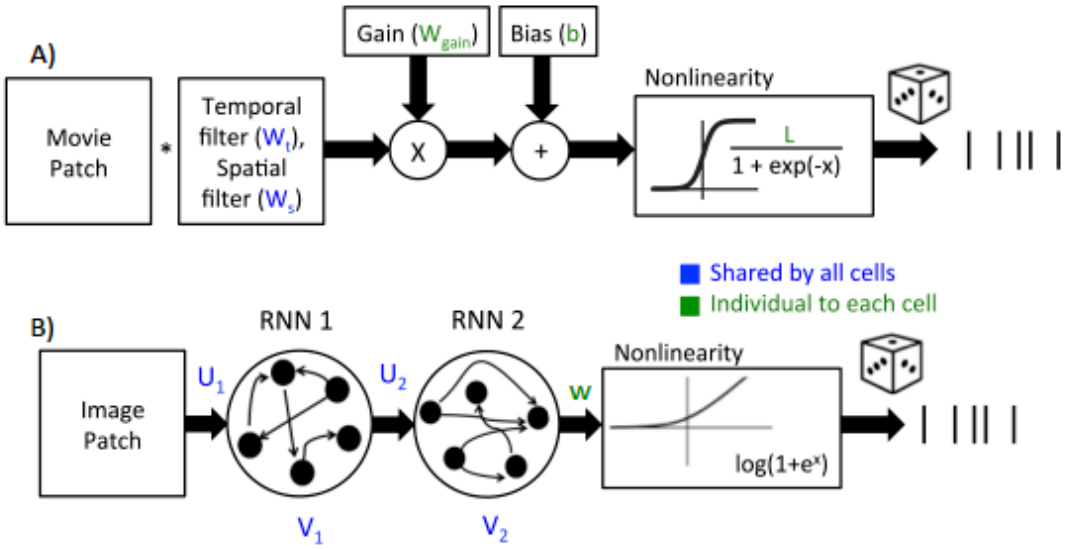
$$E(Y_i) = \lambda_i \qquad \text{var}(Y_i) = \lambda_i$$

So our variance function is

$$V(\mu_i) = \mu_i$$

Our link function must map from $(0, \infty) \to (-\infty, \infty)$. A natural choice is

$$g(\mu_i) = \log(\mu_i)$$



Training using spiking response found in retina of awake macaque monkey presented with natural image stimulus

Figure 1: Example model architectures. (A) Shared LN model. The past few frames of the stimulus images are presented as inputs which are spatiotemporally filtered and passed through a nonlinearity to produce a firing rate, which drives a Poisson spiking process. (B) Two-layer RNN. The current frame of the stimulus feeds into a sequence of RNN layers (history dependence is implicit in the hidden unit activations) and a Poisson GLM draws weighted inputs from the activations of the hidden units of the last RNN layer and outputs predicted spike trains. Thus the last RNN layer represents a shared feature pool that all the RGCs can draw from.

Observations

- Using deep networks to model neural spiking responses can significantly improve prediction over current state-of the-art models
- Exploration of deep learning can provide better insight about neuro-physiological processes

Designing the Network

Finding the Correct Configuration

- The goal of model selection is then, usually, to find the value of the hyper-parameter so that validation error is optimised.
- Essentially search in the space of all possible configurations using information about current hyper-parameters and performance index
- Global minima is a function of the configuration
- These black-box methods have two drawbacks
  - To obtain each value of c , they must execute a full network training run. Each run can take days or even weeks on many cores or multiple GPUs.
  - They do not exploit opportunities to improve the value of c further by altering   during each training run.

Emerging Paradigm

- Dynamically and automatically shrink and expand the network as needed to select a good network size during a single training run.
- Further, by altering network size during training, we can achieve a higher accuracy for the network ultimately chosen than if that network hadnbeen trained from scratch

Parametric Network

For the purpose of this section, we define a parametric neural network as a function $f(x) = \sigma_L.(\sigma_{L-1}.(..\sigma_2.(\sigma_1.(xW_1)W_2)..)W_L)$ of a d_0-dimensional row vector x, where $W_l \in \mathbb{R}^{d_{l-1}*d_l}, 1 \leq l \leq L$ are dense weight matrices of fixed dimension and $f_l : \mathbb{R} \to \mathbb{R}, 1 \leq l \leq L$ are fixed non-linear transformations that are applied elementwise, as signified by the $.()$ operator. The number of layers L is also fixed. Further, the weight matrices are trained by solving the following minimization problem: $\min_{\mathbf{W}=(W)_l} \frac{1}{|D|} \sum_{(x,y)\in D} e(f(x, \mathbf{W}), y) + \Omega(\mathbf{W})$, where D is the dataset, e is an error function that consumes a vector of fixed length d_L and the label y, and Ω is the regularizer.

Non-parametric Network

We define a nonparametric neural network in the same way, except that the dimensionality of the weight matrices is undetermined. Hence, the optimization problem becomes:

$$\min_{\mathbf{d}=(d)_l, d_l \in \mathbb{Z}_+, 1 \leq l \leq L-1} \; \min_{\mathbf{W}=(W)_l, W_l \in \mathbb{R}^{d_{l-1} * d_l}, 1 \leq l \leq L} \frac{1}{|D|} \sum_{(x,y) \in D} e(f(\mathbf{W}, x), y) + \Omega(\mathbf{W}) \quad (1)$$

Note that the dimensions d_0 and d_L are fixed because the data and the error function e are fixed. The parameter value now takes the form of a pair $(\mathbf{d}, \mathbf{W})$.

Regularizer

In the nonparametric setting, because the existence of a global minimum is not guaranteed, we may be able to reduce the error further and further by using larger and larger networks. This would be problematic, because as networks become better and better with regards to the objective, they would become more and more undesirable in practice. It turns out that in an important case, this degeneration does not occur. Define the *fan-in regularizer* Ω_{in} and the *fan-out regularizer* Ω_{out} as follows:

$$\Omega_{in}(\mathbf{W}, \lambda, p) = \lambda \sum_{l=1}^{L} \sum_{j=1}^{d_l} ||[W_l(1,j), W_l(2,j), .., W_l(d_{l-1}, j)]||_p \quad (2)$$

$$\Omega_{out}(\mathbf{W}, \lambda, p) = \lambda \sum_{l=1}^{L} \sum_{i=1}^{d_{l-1}} ||[W_l(i,1), W_l(i,2), .., W_l(i, d_l)]||_p \quad (3)$$

Observations

- How do we formulate the problem for other network architectures?
- Can we have PAC solution?

Deep Learning in Non-Stationary Environment

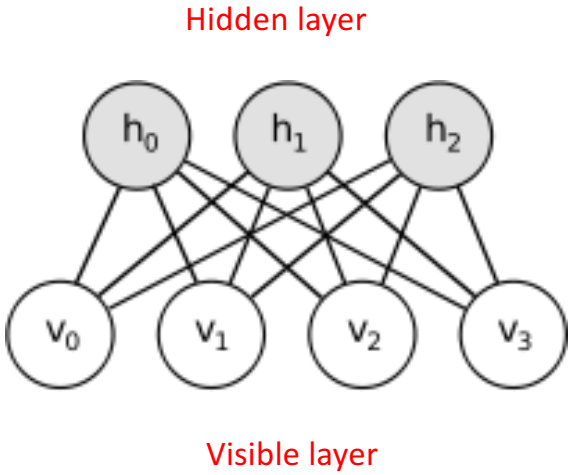
Non-stationary Environment

- The analysis of non-stationary (e.g., concept drift) streams of data involves specific issues connected with the temporal and changing nature of the data.
- Adaptive Deep Belief Networks ?
- How deep learning can be generalized to learn online from changing streams of data?
- Can deep generative models be harnessed for this purpose?

Restricted Boltzmann Machines

- Learning generative model of input data that has one layer of latent variables
- Bipartite graph with restricted connectivity
- Governed by energy function which are determined by weights linearly
- Energy function determines probabilities of networks adopting particular states (probabilities are exponential function of energies)
- By manipulating energies of joint configurations, we manipulate the probabilities the model assigns to the visible vector.
- Alternate Gibbs sampling to learn the weights of an RBM.
- Persistence Contrastive Divergence: relies on single Markov chain as a Markov chain is able to catch up with the model if parameter updates are small in comparison to mixing rate

Hinton 2006



Deep Belief Networks

- Deep Belief Networks are probabilistic models that are usually trained in an unsupervised, greedy manner.
- The basic building block of a DBN is the Restricted Boltzmann Machine (RBM) that denes a joint distribution over the inputs and binarylatent variables using undirected edges between them.
- A DBN is created by repeatedly training RBMs stacked one on top of the previous one, such that the latent variables of the previous RBM are used as data for the next one.
- The resulting DBN includes both generative connections for modeling the inputs, and recognition connections for classification
- DBNs typically require the presence of a static dataset.
  - To deal with changing streams, the usage of DBNs would require storing all the previous observations to re-train the DBN.

Adaptive Deep Belief Networks

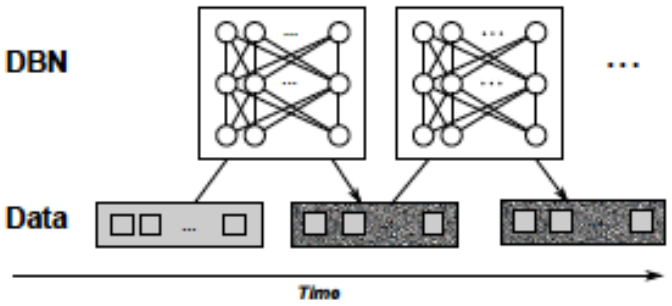


Fig. 1: Regenerative Chaining: Alternately learning a DBN from data and generating data from a DBN.

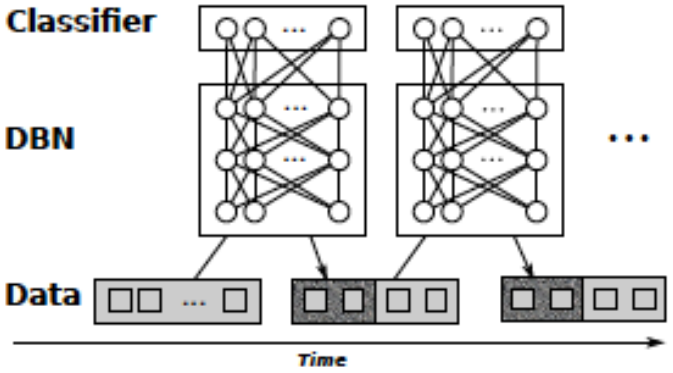


Fig. 2: ADBN: DBN+classifier are trained from both the generated samples and newly incoming data.

Comments

- Generative Models provides an opportunity to address non-stationary environment in an effective way
- We require deeper understanding of the network dynamics
- Possibly most challenging but practical problem

Efficient Deep Learning

CNN on Mobile

- Over the course of three years, CNNs have revolutionized computer vision, setting new performance standards in many important applications.
- The breakthrough has been made possible by the abundance of training data,
- the deployment of new computational hardware (most notably, GPUs and CPU clusters) and large models.
  - These models typically require a huge number of parameters (107 109) to achieve state of-the-art performance, and may take weeks to train even with high-end GPUs.
- There is a growing interest in deploying CNNs to low-end mobile devices. On such processors, the computational cost of applying the model becomes problematic, let alone training one, especially when real-time operation is needed.
- Storage of millions of parameters also complicates the deployment.
- Modern CNNs would find many more applications if both the computational cost and the storage requirement could be significantly reduced.

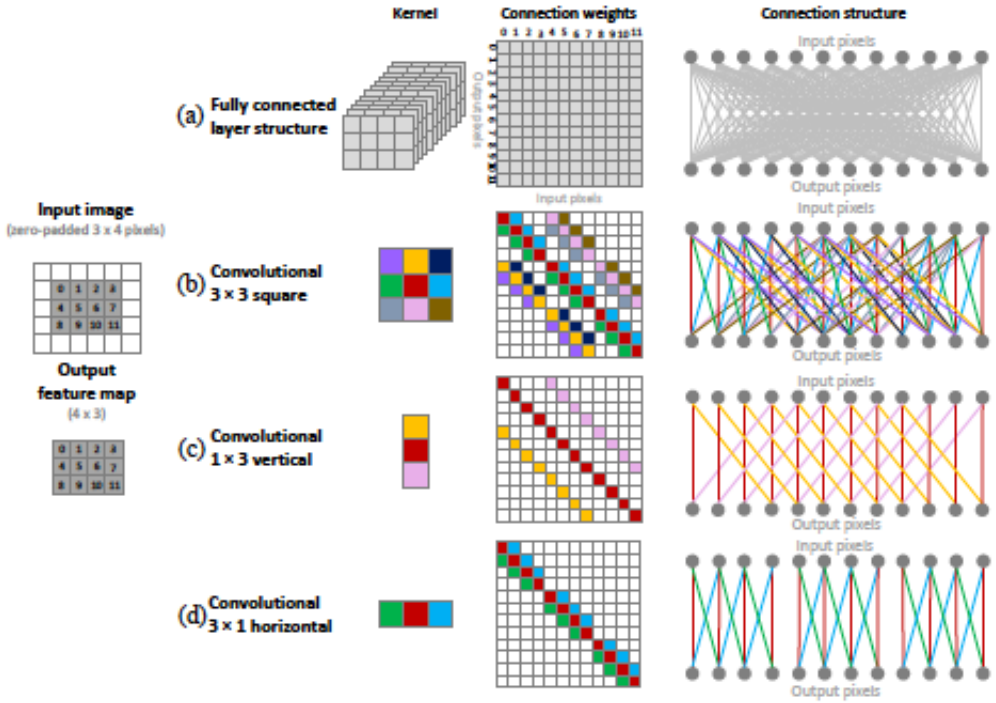


Figure 1: **Network connection structure for convolutional layers.** An input image is transformed in one layer of an neural network into an output image of same size. Connection weight maps show pairwise dependencies between input and output pixels. In (a), each node is connected to all input pixels. For (b,c,d), output pixels depend only on a subset of input pixels (shared weights are represented by unique colours). Note that sparsity increases from (a) to (d), opening up potentially more efficient implementation.

Convolutional Kernel

Formally, a convolutional kernel in a CNN is a 4D tensor $\mathcal{W} \in \mathbb{R}^{N \times d \times d \times C}$, where N, C are the numbers of the output and input feature maps respectively and d is the spatial kernel size. We also view $\mathcal{W}$ as an 3D filter array and use notation $\mathcal{W}_n \in \mathbb{R}^{d \times d \times C}$ to represent the n-th filter. Let $\mathcal{Z} \in \mathbb{R}^{X \times Y \times C}$ be the input feature map. The output feature map $\mathcal{F} = (\mathcal{F}_1, \cdots, \mathcal{F}_N)$ is defined as

$$\mathcal{F}_n(x, y) = \sum_{i=1}^{C} \sum_{x'=1}^{X} \sum_{y'=1}^{Y} \mathcal{Z}^c(x', y') \mathcal{W}_n^c(x - x', y - y'),$$

where the superscript is the index of the channels.

Approximation

The goal is to find an approximation $\tilde{\mathcal{W}}$ of $\mathcal{W}$ that facilitates more efficient computation while maintaining the classification accuracy of the CNN. We propose the following scheme:

$$\tilde{\mathcal{W}}_n^c = \sum_{k=1}^{K} \mathcal{H}_n^k (\mathcal{V}_k^c)^T, \tag{1}$$

where K is a hyper-parameter controlling the rank, $\mathcal{H} \in \mathbb{R}^{N \times 1 \times d \times K}$ is the horizontal filter, $\mathcal{V} \in \mathbb{R}^{K \times d \times 1 \times C}$ is the vertical filter (we have slightly abused the notations to make them concise, $\mathcal{H}_n^k$ and $\mathcal{V}_k^c$ are both vectors in $\mathbb{R}^d$). Both $\mathcal{H}$ and $\mathcal{V}$ are learnable parameters.

With this form, the convolution becomes:

$$\tilde{\mathcal{W}}_n * \mathcal{Z} = \sum_{c=1}^{C} \sum_{k=1}^{K} \mathcal{H}_n^k (\mathcal{V}_k^c)^T * \mathcal{Z}^c = \sum_{k=1}^{K} \mathcal{H}_n^k * \left(\sum_{c=1}^{C} \mathcal{V}_k^c * \mathcal{Z}^c \right)$$

Approximation

Based on the approximation criterion introduced in the previous section, the objective function to be minimized is:

$$(P1) \qquad E_1(\mathcal{H}, \mathcal{V}) := \sum_{n,c} \|\mathcal{W}_n^c - \sum_{k=1}^{K} \mathcal{H}_n^k (\mathcal{V}_k^c)^T\|_F^2. \tag{3}$$

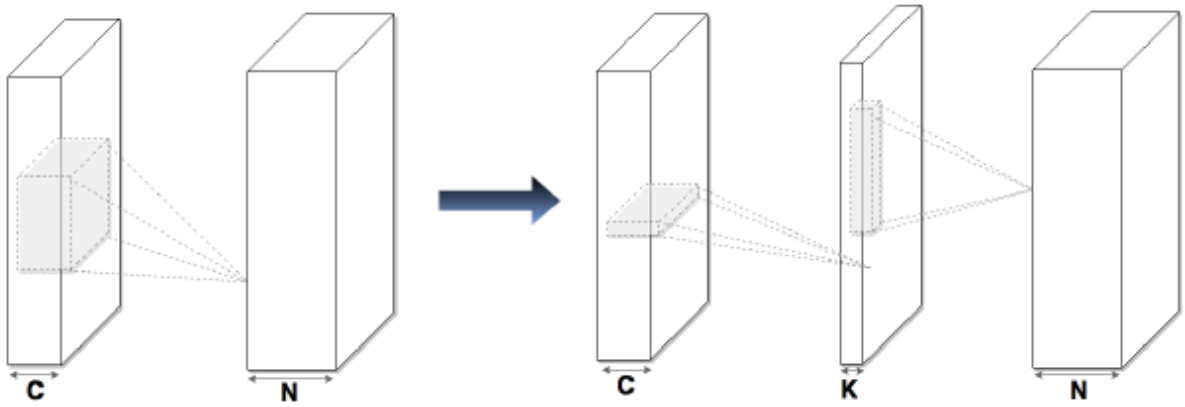


Figure 2: The proposed parametrization for low-rank regularization. Left: The original convolutional layer. Right: low-rank constraint convolutional layer with rank-K.

Comments

- Mapping Low Rank implementation to multi-core processors ?
- FPGA based accelerators?
- Deep Learning Everywhere – IOT with Deep Learning