Neural Nets

David Barber

Learning Objective

Lectures

- Motivation for neural networks.
- Understanding of feedforward networks.
- How to train neural networks.
- Autoencoders for compression.

NN background

Neural Networks

- “Neural Nets” traditionally refers to layered information processing models.
- Inspiration comes from biology in which information is typically processed in layers and hierarchically represented.
- We will focus here on traditional ‘feedforward’ models.
- NNs can be problematic to train. However, once trained they are *lightening fast* to use on a novel datapoint.

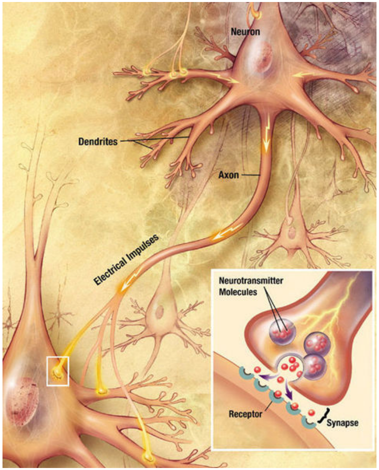
Deep Learning

- NNs have resurged in interest in the last few years. Also called ‘deep learning’.
- Sense that very complex tasks (object recognition, learning complex structure in data) requires going beyond simple statistical techniques.
- Recent results using NNs are encouraging.
- Big Interest (!) in this – Google, Facebook, *etc.*

Astonishing Hypothesis: Crick

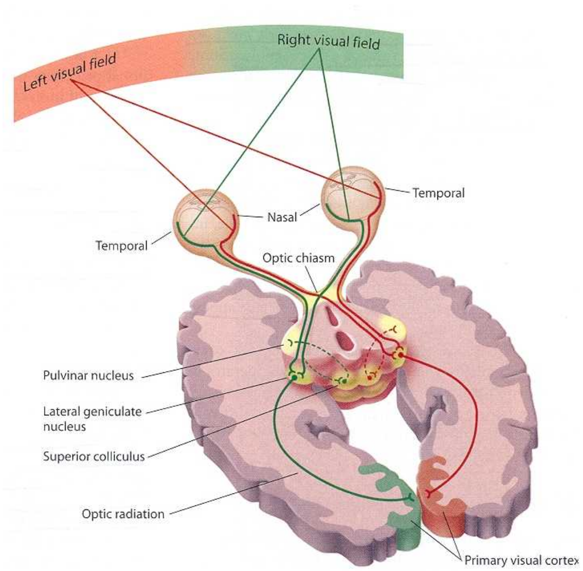
"A person's mental activities are entirely due to the behaviour of nerve cells and the molecules that make them up and influence them."

Neurons

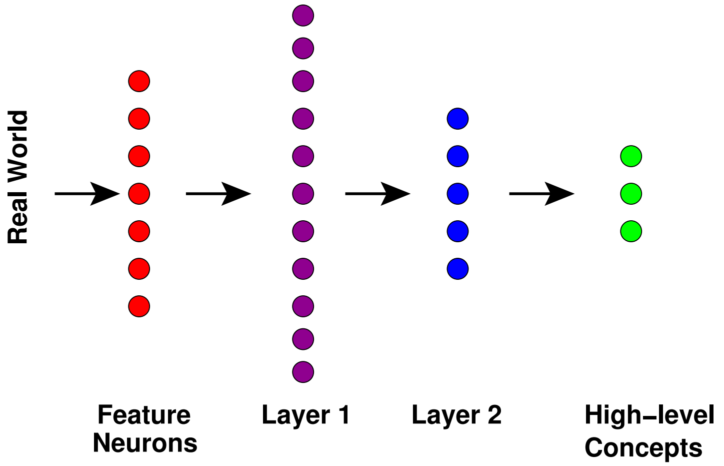


Learning = adjusting the strengths of the connections between neurons.

Visual Pathway



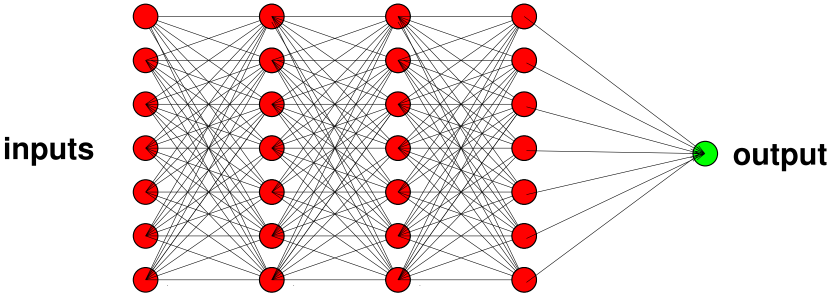
Information Processing in Brains



Hierarchical; Modular; Binary; Parallel; Noisy

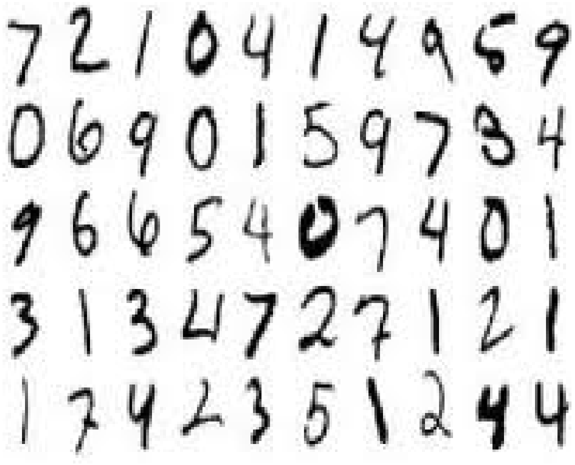
Connectionism

- 1960 Realised a perceptron can only solve simple tasks.
- 1970 Decline in interest.
- 1980 New computing power made training multilayer networks feasible.



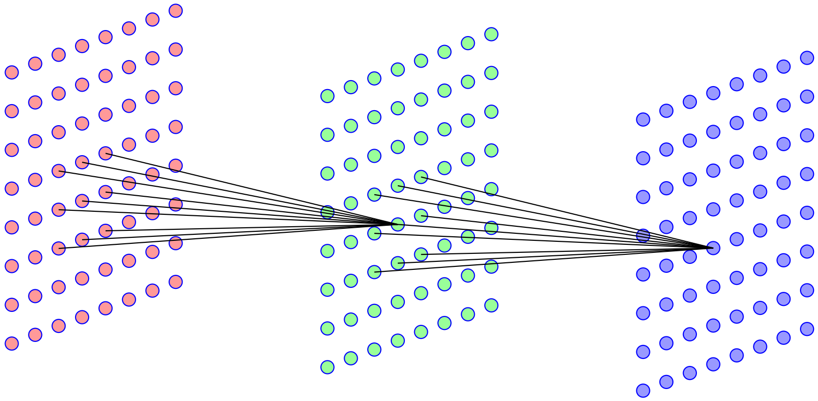
Each node (or ‘neuron’) computes a function of a weighted combination of parental nodes: $h_j = \sigma(\sum_i w_{ij} h_i)$

Digit Recognition using a Neural Network

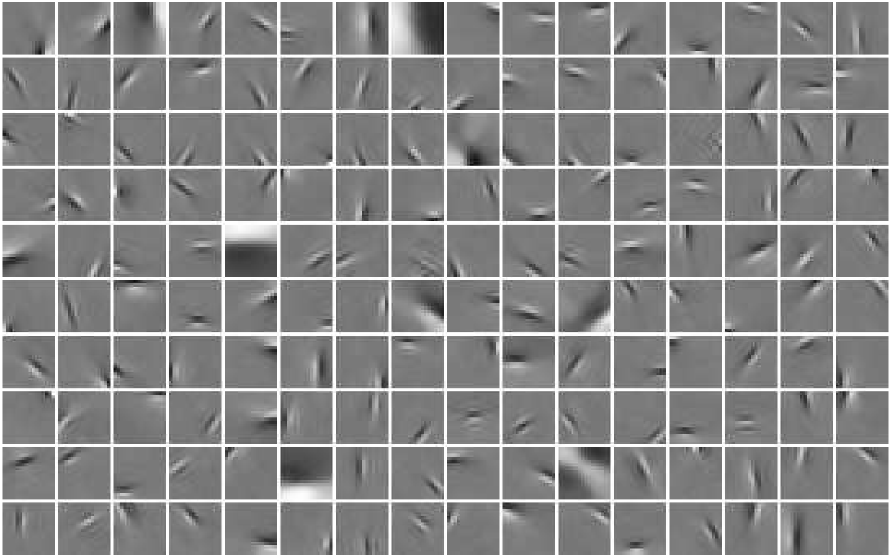


Computer makes error less than $1/1000$.

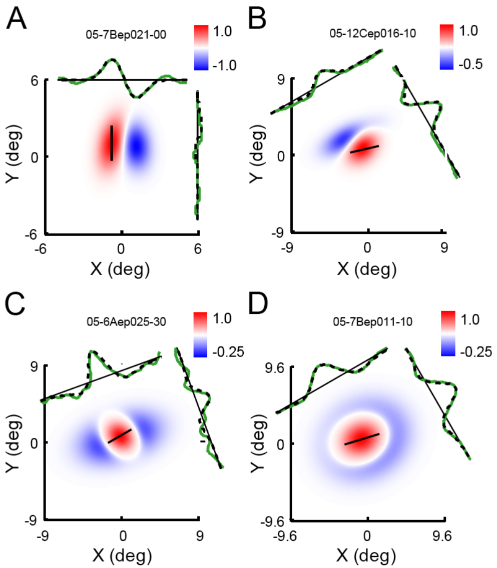
Digit recognition using a neural network



Feature neurons in the first layer



Feature neurons in a real brain



Definition

- Consider a vector input $\mathbf{x}$ that gets mapped to an output $\mathbf{y}$ through intermediate layers.
- For a network with L layers, we write the vector function that the network computes as

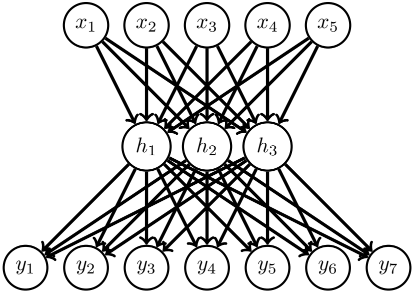
  $$f(\mathbf{x}|\mathcal{W}) \equiv \sigma_L\left(\mathbf{W}_L \mathbf{h}_{L-1}\right)$$

  where

  $$\mathbf{h}_l = \sigma_l\left(\mathbf{W}_l \mathbf{h}_{l-1}\right), \qquad l = 2, \ldots, L-1, \qquad \mathbf{h}_1 = \sigma_1\left(\mathbf{W}_1 \mathbf{x}\right)$$

- The dimension of each hidden layer is given by $H_l \equiv \dim(\mathbf{h}_l)$.
- The transfer functions $\sigma_l(x)$ can be different for each layer (or even different for units within the same layer).
- Common also to include 'bias' terms $\mathbf{b}$, so that $\mathbf{h}_l = \sigma_l\left(\mathbf{W}_l \mathbf{h}_{l-1} + \mathbf{b}_l\right)$. The biases form part of the parameter set.
- Typical to use an invertible monotonic function, $\sigma(x) = e^x/(1+e^x)$ or $\sigma(x) = \tanh(x)$.

The NN diagram



- Input-output neural network with a single hidden layer.
- Can write this more compactly as $\mathbf{x} \to \mathbf{h} \to \mathbf{y}$.
- Can have more than one hidden layer: $\mathbf{x} \to \mathbf{h}_1 \to \mathbf{h}_2 \ldots \to \mathbf{y}$.
- This is a diagrammatic representation of a function $\mathbf{y} = f(\mathbf{x}|\mathcal{W})$.
- If the transfer functions are all linear, the function is linear and additional hidden layers play no role. Typically we are interested in the situation of using non-linear transfer functions – takes us away from classical statistical techniques.
- Very rich functional structure but hard to analyse.

Regression

- Neural networks can be used to solve a regression problem by minimising the squared loss on a set of input-output example pairs, $(\mathbf{x}^n, \mathbf{y}^n)$, $n = 1, \ldots, N$:

$$E(\mathcal{W}) = \sum_{n=1}^{N} \left[\mathbf{y}^n - \mathbf{f}(\mathbf{x}^{\mathbf{n}}|\mathcal{W})\right]^2$$

- Optimisation can be achieved using gradient descent (slow) or higher order methods such as BFGS or conjugate gradients.
- A well documented issue with training 'deep' networks is that it is difficult to find a good initialisation of the weights $\mathcal{W}$.
- For this reason, 'layerwise' training algorithms and unsupervised-initialisation approaches have been popular.
- Common to use distributed computing methods on large scale problems.
- Then use Stochastic Gradient Descent, which is robust to processor synchronisation issues.
- Note that the gradient can be computed efficiently using the 'back-propagation' algorithm, or more generally just using AutoDiff.

Loss functions

Regression

Squared loss $E(\mathbf{y}, \mathbf{h}^L) = \left(\mathbf{y} - \mathbf{h}^L\right)^2$ is the most common.

---

classification

Logistic loss for binary classification $y \in \{+1, -1\}$. We have a single output with $\sigma_L(x) \equiv \sigma(x)$, where $\sigma(x)$ is the logistic sigmoid.

$$E(y, h^L) = -\log\left(\mathbb{I}\left[y = +1\right] h^L + \mathbb{I}\left[y = -1\right](1 - h^L)\right)$$

Softmax for multiple classes. Have C outputs, one for each class with:

$$\sigma_L(x_c) \equiv \frac{e^{W_c^L x_c}}{\sum_c e^{W_c^L x_c}}$$

$$E(\mathbf{y}, \mathbf{h}^L) = -\log \sum_{c=1}^{C} \mathbb{I}\left[y = c\right] h_c^L$$

Penalty Terms

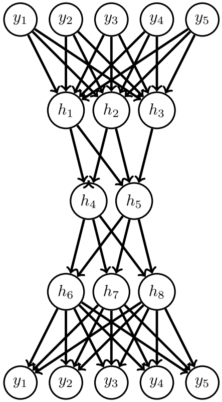
- NNs are potentially very powerful functions.
- Given enough hidden layers, pretty much any function (including random noise!) can be modelled.
- To discourage overfitting, common to add penalty terms to the error.
- The most common regularisation term is

  $$\lambda \sum_{l} \sum_{ij} \left(W_{ij}^{l}\right)^2$$

  Easy to include this term in the gradient calculation.
- Note that usually no need to penalise bias terms.
- Regularisation parameter λ is set typically by cross validation.

Specific NN Architectures

Autoencoder



- Try to map the input back to itself.
- Hidden Layer with the smallest number of units is called the ‘bottleneck’.
- The bottleneck forces the network to try to find a low dimensional representation of the data.
- Useful for unsupervised learning.

Autoencoder

- Aim is to try to find a lower dimensional ('compressed') representation of higher dimensional data $\mathbf{y}$.
- When the output is set to the input, the objective is to mimimise

$$E(\mathcal{W}) = \sum_{n=1}^{N} \left[\mathbf{y}^n - \mathbf{f}(\mathbf{y^n}|\mathcal{W})\right]^2$$

  wrt $\mathcal{W}$.
- Typically use several hidden layers, $\mathbf{y} \rightarrow \mathbf{h}^1 \rightarrow \ldots \rightarrow \mathbf{h}^L$.
- The 'bottleneck' (hidden layer with smallest dimension) effectively means that the outputs $\mathbf{y}$ are then approximated by the lower dimensional representations in the bottleneck layer.
- These representations can be used for low dimensional encodings of the data $\mathbf{y}$ or for subsequent processing (such as classification).

`demoAutoencoderMNIST.m`

Autoencoder on MNIST digits

- 60,000 training images ($28 \times 28 = 784$ pixels).
- Use a form of autoencoder to find a lower (30) dimensional representation.

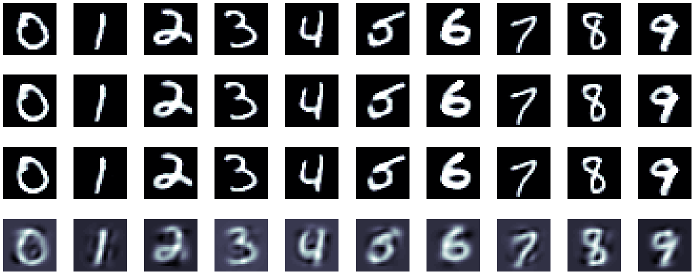
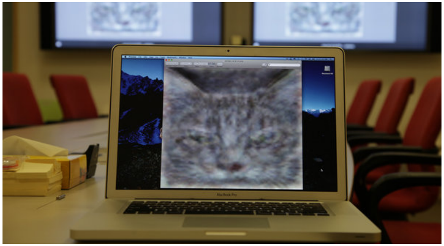
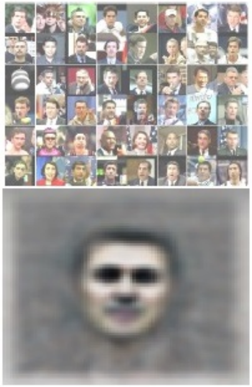


Figure: Reconstructions using $H = 30$ components. From the Top: Original image, NN using 1 layer (reconstruction error=2.42); NN using $H = [30, 100]$ (reconstruction error=2.38); PCA (reconstruction error=14.46)

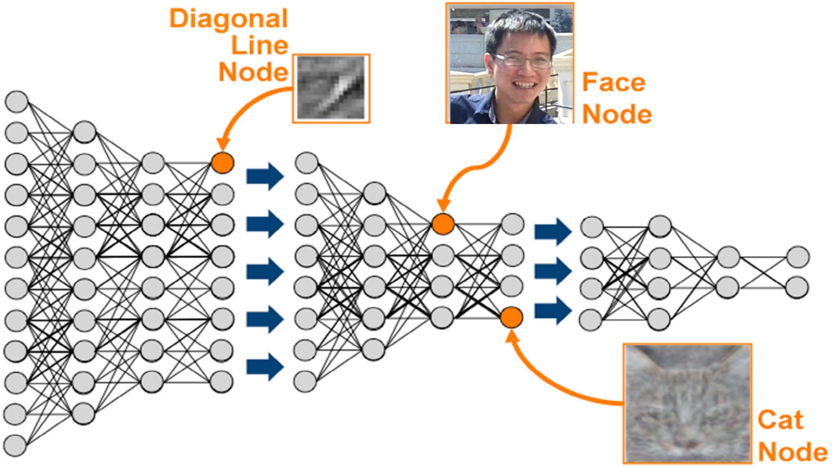
Google Cats

- 10 Million Youtube video frames (200x200 pixel images).
- Use a specialised autoencoder with 9 layers (1 billion weights).
- 2000 computers + two weeks of computing.
- Examine bottleneck units to see what images they most respond to.





Google Autoencoder



From Nando De Freitas

Autoencoders and PCA

PCA

- Given a set of data $Y = \left(\mathbf{y}^1, \ldots, \mathbf{y}^N\right)$ consider trying to find a rank H approximation $\tilde{Y}$ to Y that minimises the squared loss $||Y - \tilde{Y}||^2$.
- Well known that the minimal rank H loss is given by taking the singular value decomposition of $Y = USV^\mathsf{T}$, taking the largest H singular values:

$$\tilde{Y} = U_H S_H V_H^\mathsf{T}$$

- This is called Principal Components Analysis and is a classical algorithm.
- There is a polynomial algorithm for SVD and also very fast approximations.

Autocencoder

- Consider $\mathbf{y} \to \mathbf{h} \to \tilde{\mathbf{y}}$, a single hidden layer net with linear transfer $\sigma_2(x) = x$. Let's call the output of the network $\tilde{Y}$. Then the squared loss is given by

$$E = ||Y - \tilde{Y}||^2$$

  where $\tilde{Y} = W_2 \sigma_1(W_1 Y)$. If $\sigma_1(x) = x$ then $\tilde{Y} = W_2 W_1 Y = W_2 W_1 USV^\mathsf{T}$ which has rank H. But this is the same as PCA by setting $W_2 = U_H$ and $W_1 = U_H^\mathsf{T}$ since then $\tilde{Y} = U_H S_H V_H^\mathsf{T}$.
- Hence PCA is equivalent to an autoencoder $\mathbf{y} \to \mathbf{h} \to \tilde{\mathbf{y}}$ with linear transfer functions.

Autoencoders and PCA

- Let's try to make a more powerful autoencoder by using a non-linear transfer function $\sigma_1(x)$.
- Then $\tilde{Y} = W_2\sigma_1(W_1Y)$.
- This means that $\tilde{Y}$ has still has rank H, the size of the hidden layer.
- Note that $\sigma_1(W_1Y)$ is also some matrix.
- Hence, overall, this corresponds to trying to find a rank H matrix $\tilde{Y}$.
- But this *optimally* solved by PCA – the non-linearity cannot improve the power of this model.
- We arrive at an important conclusion, that for a linear output $\sigma_2(x) = x$, the optimal transfer function on the single-hidden layer autoencoder is given by $\sigma_1(x) = x$, with the weights set by the SVD (PCA) of Y.
- In other words, you cannot beat PCA if you only have a linear transfer function at the output (for a single hidden layer autoencoder).

Autoencoders

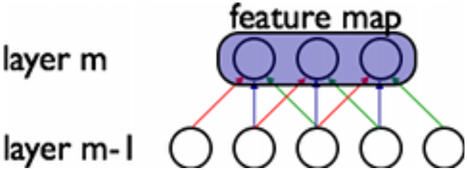
To go beyond PCA

- We need a non-linear output function
- If the output function is linear, and we have only a single hidden layer, we cannot beat PCA.

Caveat Emptor

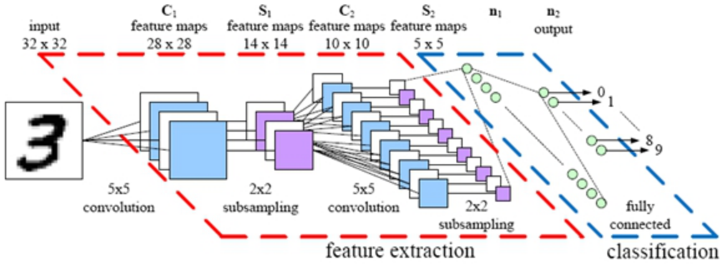
- Consider a situation in which we have N units in the bottom layer. Unit i will receive activation a_i^n for datapoint $\mathbf{x}^n$.
- Define the value of the bottom hidden layer to be $\sigma_{L-1}(x_i) = \mathbb{I}\left[x_i - a_i^i\right]$ (this can be effectively achieved using a sigmoid function). This means that when vector $\mathbf{x}^n$ is inputted, bottom hidden unit n is 1 and all others are zero.
- We can then perfectly reconstruct $\mathbf{x}^n$ by using a linear output $\sigma_L(x) = x$ with weights coming out of hidden unit n to be equal to $\mathbf{x}^n$.
- Hence, even for a bottleneck with dimension 1, we can perfectly reconstruct the input for N training patterns provided that we have at least N units in the bottom hidden layer.

Convolutional NNs



- Consider that the input to the net is an image $\mathbf{x}$ and that we wish to detect if there is a bicycle in the image.
- We want the detector to have translation invariance, so that no matter where the bicycle is in the image, the net will recognise it.
- We can then imagine a small ‘bicycle detector’ node that will look at a local set of pixels in the image and respond strongly if it sees a bicycle.
- We then replicate this detector across all positions in the image.
- This generates the ‘feature’ map in the above diagram in which weights with the same colour are shared.
- We can then add an additional ‘max pooling’ layer on top of this so that if any of the feature map neurons responds strongly, then we know there is a bicycle somewhere in the image. This can be considered a form of ‘sub-sampling’.

Convolutional NNs



- CNNs are particularly popular in image processing
- Often the feature maps correspond (not to macro features such as bicycles) but micro features.
- For example, in handwritten digit recognition they correspond to small constituent parts of the digits.
- These are used then to process the image into a representation that is better for recognition.

NNs in NLP

Bag of Words

- We have D words in a dictionary, `aardvark`,…,`zorro` so that we can relate each word with it's dictionary index.
- We can also think of this as a Euclidean embedding $\mathbf{e}$:

$$\texttt{aardvark} \to \mathbf{e}_{\texttt{aardvark}} = \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}, \qquad \texttt{zorro} \to \mathbf{e}_{\texttt{zorro}} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 1 \end{pmatrix}$$

Word Embeddings

- Idea is to replace the Euclidean embeddings $\mathbf{e}$ with embeddings (vectors) $\mathbf{v}$ that are *learned*.
- Objective is, for example, next word prediction accuracy.
- These are often called 'neural language models'.

NNs in NLP

- Each word w in the dictionary has an associated embedding vector $\mathbf{v}_w$. Usually around 200 dimensional vectors are used.
- Consider the sentence:

  `the cat sat on the mat`

  and that we wish to predict the word `on` given the two preceding `cat sat` and two succeeding words `the mat`
- We can use a network that has inputs $\mathbf{v}_{\texttt{cat}}$, $\mathbf{v}_{\texttt{sat}}$, $\mathbf{v}_{\texttt{the}}$, $\mathbf{v}_{\texttt{mat}}$
- The output of the network is a probability over all words in the dictionary $p(w|\{\mathbf{v}_{inputs}\})$.
- We want $p(w = \texttt{on}|\mathbf{v}_{\texttt{cat}}, \mathbf{v}_{\texttt{sat}}, \mathbf{v}_{\texttt{the}}, \mathbf{v}_{\texttt{mat}})$ to be high.
- The overall objective is then to learn all the word embeddings and network parameters subject to predicting the word correctly based on the context.
- Note that one can also think about this as $\mathbf{v}_w = \mathbf{E}\mathbf{e}_w$ where the embedding vectors are the rows of the embedding matrix $\mathbf{E}$. Thus this is a NN with Euclidean (one-of-D) inputs and first layer matrix $\mathbf{E}$.

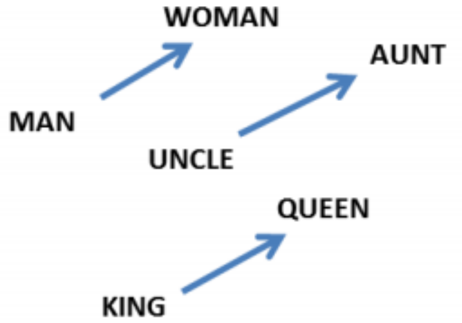
Word Embeddings

FRANCE	JESUS	XBOX	REDDISH	SCRATCHED	MEGABITS
AUSTRIA	GOD	AMIGA	GREENISH	NAILED	OCTETS
BELGIUM	SATI	PLAYSTATION	BLUISH	SMASHED	MB/S
GERMANY	CHRIST	MSX	PINKISH	PUNCHED	BIT/S
ITALY	SATAN	IPOD	PURPLISH	POPPED	BAUD
GREECE	KALI	SEGA	BROWNISH	CRIMPED	CARATS
SWEDEN	INDRA	PSNUMBER	GREYISH	SCRAPED	KBIT/S
NORWAY	VISHNU	HD	GRAYISH	SCREWED	MEGAHERTZ
EUROPE	ANANDA	DREAMCAST	WHITISH	SECTIONED	MEGAPIXELS
HUNGARY	PARVATI	GEFORCE	SILVERY	SLASHED	GBIT/S
SWITZERLAND	GRACE	CAPCOM	YELLOWISH	RIPPED	AMPERES

Given a word (`France`, for example) we can find which words w have embedding vectors closest to $\mathbf{v}_{\texttt{France}}$. From Ronan Collabert (2011).

Word Embeddings

There appears to be a natural 'geometry' to the embeddings. For example, there are directions that correspond to gender.

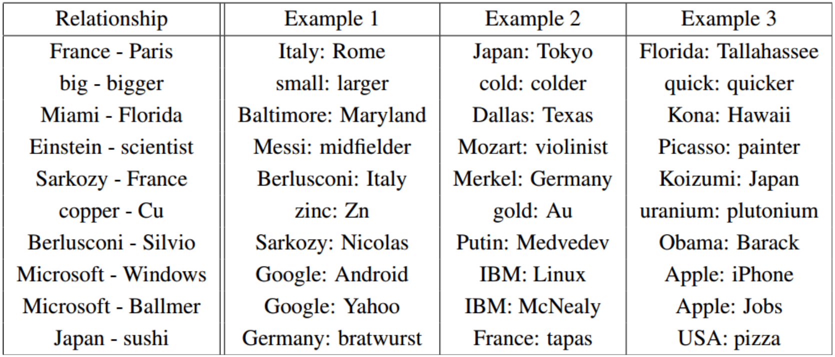


$$\mathbf{v}_{\texttt{woman}} - \mathbf{v}_{\texttt{man}} \approx \mathbf{v}_{\texttt{aunt}} - \mathbf{v}_{\texttt{uncle}}$$

$$\mathbf{v}_{\texttt{woman}} - \mathbf{v}_{\texttt{man}} \approx \mathbf{v}_{\texttt{queen}} - \mathbf{v}_{\texttt{king}}$$

From Mikolov (2013).

Word Embeddings: Analogies



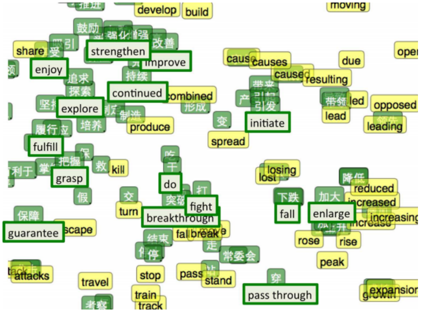
Relationship	Example 1	Example 2	Example 3
France - Paris	Italy: Rome	Japan: Tokyo	Florida: Tallahassee
big - bigger	small: larger	cold: colder	quick: quicker
Miami - Florida	Baltimore: Maryland	Dallas: Texas	Kona: Hawaii
Einstein - scientist	Messi: midfielder	Mozart: violinist	Picasso: painter
Sarkozy - France	Berlusconi: Italy	Merkel: Germany	Koizumi: Japan
copper - Cu	zinc: Zn	gold: Au	uranium: plutonium
Berlusconi - Silvio	Sarkozy: Nicolas	Putin: Medvedev	Obama: Barack
Microsoft - Windows	Google: Android	IBM: Linux	Apple: iPhone
Microsoft - Ballmer	Google: Yahoo	IBM: McNealy	Apple: Jobs
Japan - sushi	Germany: bratwurst	France: tapas	USA: pizza

Given a relationship, `France-Paris`, we get the 'relationship' embedding

$$\mathbf{v} = \mathbf{v}_{\texttt{Paris}} - \mathbf{v}_{\texttt{France}}$$

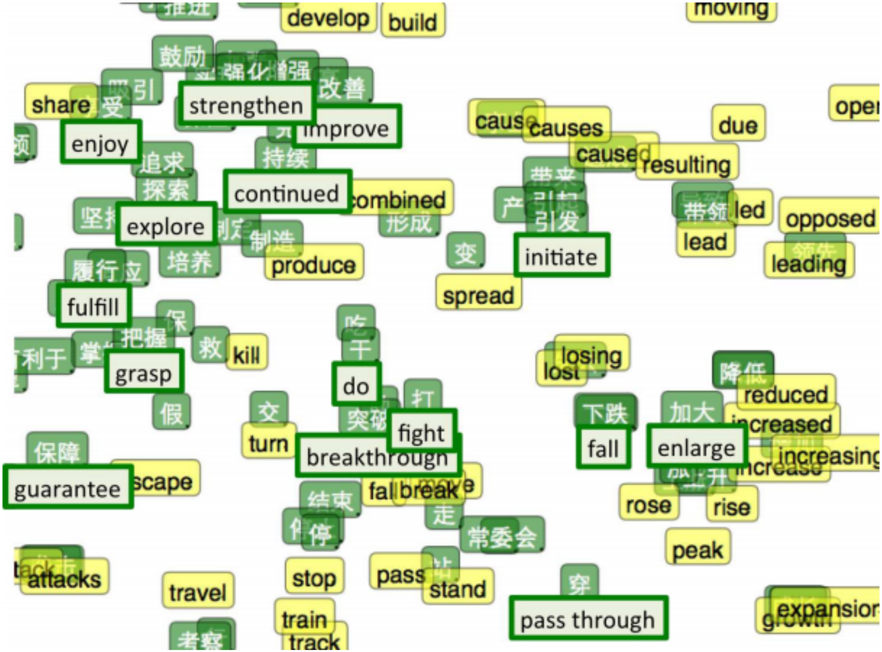
Given `Italy` we can calculate $\mathbf{v}_{\texttt{Italy}} + \mathbf{v}$ and find the word in the dictionary which has closest embedding to this (it turns out to be `Rome`!). From Mikolov (2013).

Word Embeddings: Constrained Embeddings

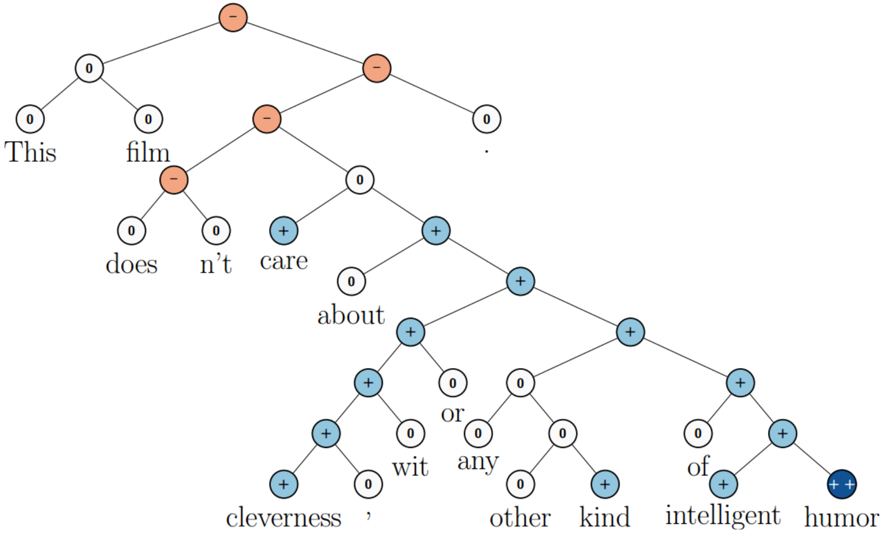


- We can learn embeddings for English words and embeddings for Chinese words.
- However, when we know that a Chinese and English word have a similar meaning, we add a constraint that the word embeddings $\mathbf{v}_{\mathtt{ChineseWord}}$ and $\mathbf{v}_{\mathtt{EnglishWord}}$ should be close.
- We have only a small amount of labelled 'similar' Chinese-English words (these are the green border boxes in the above; they are standard translations of the corresponding Chinese character).
- We can visualise in 2D (using t-SNE) the embedding vectors. See Socher (2013)

Word Embeddings: Constrained Embeddings

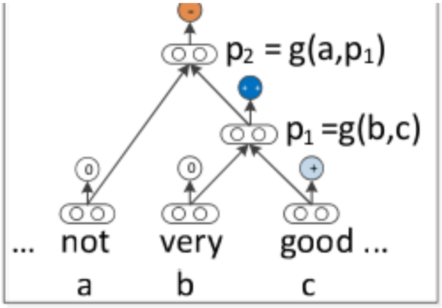


Recursive Nets and Embeddings



Stanford Sentiment Treebank. Consists of parsed sentences with sentiment labels $(--, -, 0, +, ++)$ for each node (phrase) in the tree. 215,000 labelled phrases (obtained from three humans).

Recursive Nets and Embeddings



- Idea is to recursively combine embeddings such that they accurately predict the sentiment at each node.

Recursive Nets and Embeddings

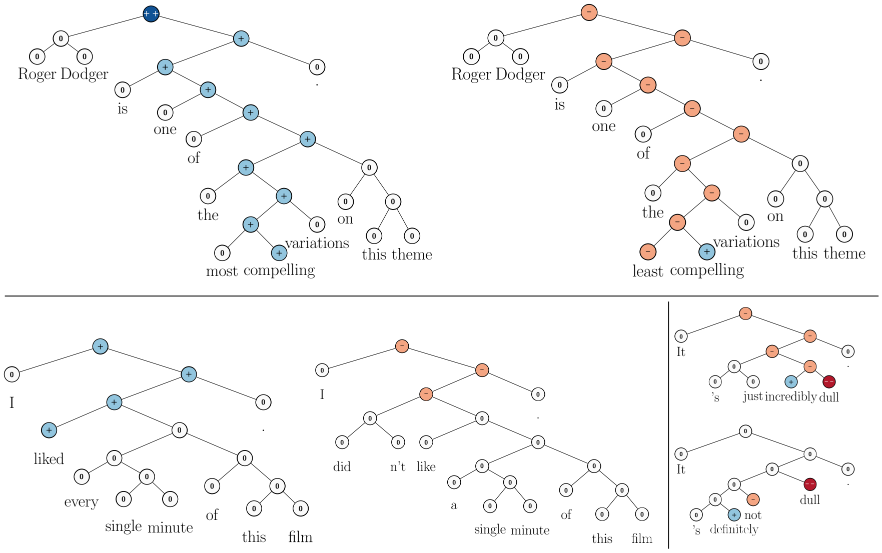
Training

- We have a softmax classifier for each node in the tree, to predict the sentiment of the phrase beneath this node in the tree.
- The weights of this classifier are shared across all nodes.
- At the leaf nodes at the bottom of the tree, the inputs to the classifiers are the word embeddings.
- The embeddings are combined by another network g with common parameters, which forms the input to the sentiment classifier.
- We then learn all the embeddings, shared classifier parameters and shared combination parameters to maximise the classification accuracy.

Prediction

- For a new movie review, the review is first parsed using a standard grammar tree parser.
- This forms the tree which can be used to recursively form the sentiment class label for the review.
- Currently the best sentiment classifier. Socher (2013)

Recursive Nets and Embeddings



Roger Dodger is one of the most compelling variations on this theme .
Roger Dodger is one of the least compelling variations on this theme .
I liked every single minute of this film .
I did n't like a single minute of this film .
It 's just incredibly dull .
It 's definitely not dull .

Figure 9: RNTN prediction of positive and negative (bottom right) sentences and their negation.

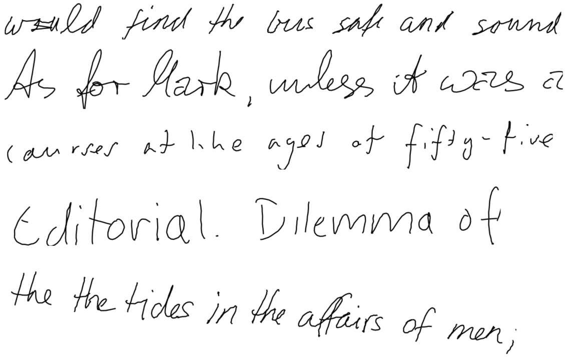
Time Series Models

Recurrent Nets

$$\begin{tikzcd}
x_1 \arrow[d, "A"'] & x_2 \arrow[d, "A"'] & x_3 \arrow[d, "A"'] \\
h_1 \arrow[r, "B"] \arrow[d, "C"'] & h_2 \arrow[r, "B"] \arrow[d, "C"'] & h_3 \arrow[d, "C"'] \\
y_1 & y_2 & y_3
\end{tikzcd}$$

- RNNs are used in timeseries applications
- The basic idea is that the hidden units at time h_t (and possibly output y_t) depend on the previous state of the network $h_{t-1}, x_{t-1}, y_{t-1}$ for inputs x_t and outputs y_t.
- In the above network, I ‘unrolled the net through time’ to give a standard NN diagram.
- I omitted the potential links from x_{t-1}, y_{t-1} to h_t.

Handwriting Generation using a RNN



would find the bus safe and sound

As for Mark, unless it was a

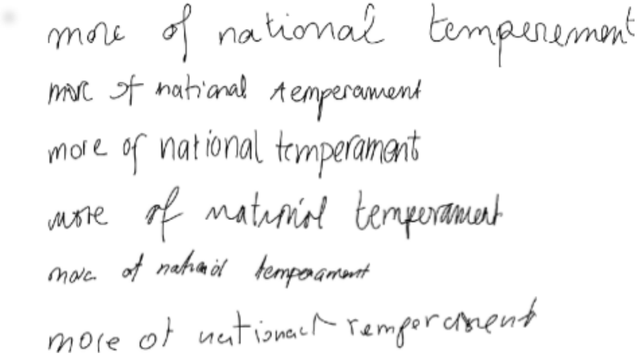
courses at like ages of fifty-five

Editorial. Dilemma of

the the tides in the affairs of men;

Some training examples.

Handwriting Generation using a RNN



Some generated examples. Top line is real handwriting, for comparison. See Alex Grave's work.

Computing the Gradient

For a squared loss, we might have an objective of the form

$$E(A, B, C) = \sum_t (y_t - f(h_t; C))^2, \qquad h_t = g(x_t, h_{t-1}; A, B)$$

- The output of the network at time t is some function f (parameterised by a matrix C) of the hidden state h_t.
- The hidden value at time t is some function g (parameterised by input to hidden weights A and hidden to hidden weights B) of the input x_t and previous hidden value h_{t-1}.
- To train a recurrent network we need to calculate the gradient with respect to A, B, C. There are several ways to do this with varying storage and time performances, see Williams and Zipser (1995) for a detailed comparison.
- We will discuss two methods below: RTRL (which is a single forward pass in time but has high storage cost) and BPTT (which is a forward and backward pass with more modest storage cost). RTRL is straightforward, but BPTT requires an understanding of parameter tying (or more generally AutoDiff).

Real Time Recurrent Learning (Williams and Zipser)

Consider a recurrent network with parameters θ and (for example) a squared loss:

$$E(\theta) = \frac{1}{2}\sum_t (y_t - f_\theta(h_t))^2, \qquad h_t = g_\theta(x_t, h_{t-1})$$

Then

$$\frac{dE}{d\theta} = -\sum_t (y_t - f_\theta(h_t))\frac{df_\theta(h_t)}{d\theta},$$

where

$$\frac{df_\theta(h_t)}{d\theta} = \frac{\partial f_\theta(h_t)}{\partial\theta} + \frac{\partial f_\theta(h_t)}{\partial h_t}\frac{dh_t}{d\theta}$$

and we can use the recursion:

$$\frac{dh_t}{d\theta} = \frac{\partial g_\theta(x_t, h_{t-1})}{\partial\theta} + \frac{\partial g_\theta(x_t, h_{t-1})}{\partial h_{t-1}}\frac{dh_{t-1}}{d\theta}$$

Hence the gradient of an RNN can be computed using a single forward pass in time. However, this has very high storage cost since for a network with H units, $\frac{dh_t}{d\theta}$ would have H^3 elements. For all but special cases, this will be impractical.

Parameter Tying

- Consider a simple objective such as

$$E(\theta) = [y - f(\theta g(x\theta))]^2$$

- As a network diagram, this would look something like this $x \underset{\theta}{\rightarrow} h \underset{\theta}{\rightarrow} y$ in which the parameters from the input to hidden layer and hidden layer to output are tied.
- We can calculate the gradient directly using the usual chain rule of calculus:

$$\frac{dE}{d\theta} = -2\,[y - f(\theta g(x\theta))]\, f'(\theta g(x\theta))\,(\theta g'(x\theta)x + g(x\theta))$$

where f' and g' denote the derivatives of f and g respectively.

Parameter Tying

Another way to do this is to consider

$$F(\theta_1, \theta_2) = \left[y - f(\theta_1 g(x\theta_2))\right]^2$$

which is a Network with unconstrained parameters, $x \underset{\theta_1}{\rightarrow} h \underset{\theta_2}{\rightarrow} y$. Then

$$\frac{dF}{d\theta} = \frac{\partial F}{\partial \theta_1}\frac{\partial \theta_1}{\partial \theta} + \frac{\partial F}{\partial \theta_2}\frac{\partial \theta_2}{\partial \theta}$$

If we now constrain $\theta_1 = \theta_2 = \theta$, we obtain

$$\frac{dE}{d\theta} = \frac{dF(\theta, \theta)}{d\theta} = \left.\frac{\partial F(\theta_1, \theta_2)}{\partial \theta_1}\right|_{\theta_1=\theta_2=\theta} + \left.\frac{\partial F(\theta_1, \theta_2)}{\partial \theta_2}\right|_{\theta_1=\theta_2=\theta}$$

where $|_{\theta_1=\theta_2=\theta}$ means that we evaluate the resulting expression (after calculating the partial derivative) at the constrained values.

Parameter Tying

We can verify that this works:

$$\frac{\partial F(\theta_1, \theta_2)}{\partial \theta_1} = -2\left[y - f(\theta_1 g(x\theta_2))\right] f'(\theta_1 g(x\theta_2)) g(x\theta_2)$$

and

$$\frac{\partial F(\theta_1, \theta_2)}{\partial \theta_2} = -2\left[y - f(\theta_1 g(x\theta_2))\right] f'(\theta_1 g(x\theta_2)) \theta_1 x g'(x\theta_2)$$

Summing these two and evaluating at $\theta_1 = \theta_2 = \theta$, we obtain the correct result.

Parameter Tying

The conclusion is that we can deal with parameter tying in any objective by the following procedure:

1. Treat all parameters as independent and calculate the gradient with respect to each independent parameter.
2. Sum all the resulting independent gradients together.
3. Evaluate the expression by setting all the independent parameters to the same value.

- Note that this is a general result and can be used to deal with parameter tying in any objective, not just Deep Learning and Neural Nets.
- (Of course, this is again just a special case of AutoDiff).

Backprop Through Time

$$\begin{tikzcd}
x_1 \arrow[d, "A"'] & x_2 \arrow[d, "A"'] & x_3 \arrow[d, "A"'] \\
h_1 \arrow[r, "B"] \arrow[d, "C"'] & h_2 \arrow[r, "B"] \arrow[d, "C"'] & h_3 \arrow[d, "C"'] \\
y_1 & y_2 & y_3
\end{tikzcd}$$

(a)

$$\begin{tikzcd}
x_1 \arrow[d, "A_1"'] & x_2 \arrow[d, "A_2"'] & x_3 \arrow[d, "A_3"'] \\
h_1 \arrow[r, "B_1"] \arrow[d, "C_1"'] & h_2 \arrow[r, "B_2"] \arrow[d, "C_2"'] & h_3 \arrow[d, "C_3"'] \\
y_1 & y_2 & y_3
\end{tikzcd}$$

(b)

Figure: **(a)**: A recurrent Neural Net here written for only 3 timesteps. **(b)**: Unconstrained version of (a) used to derive BPTT.

Backprop Through Time

- Perhaps the most obvious approach is to directly calculate the gradient by a forward-propagation algorithm (RTRL).
- A more common approach is Backprop Through Time (BPTT).
- Given our understanding about parameter tying, we see that we can calculate the gradient of the recurrent NN objective by first treating all parameters as independent, and then running standard backprop on the resulting architecture.
- Finally, we simply sum all the corresponding gradients (for example with respect to the matrices A_1, A_2, A_3 to calculate the gradient with respect to A) and subsequently set the parameters to be equal.
- This is an efficient exact algorithm which, in contrast to RTRL, runs backwards in time.
- Despite having efficient algorithms to compute the gradient, training RNNs is considered particularly challenging and either specialised optimisation procedures, or modifications to the standard architecture (such as LSTM), are usually required.

Generative Models

Generative Models

- Generative models parameterise a joint distribution in the form $p(v, h) = p(v|h)p(h)$ where v are observable 'visible' variables and h latent 'hidden' variables.
- The term $p(v|h)$ expresses how the observations in the world are generated according to known 'laws'. Generative models are very common in the natural sciences.
- From a model, we can then form $p(h|v)$ to understand what latent state likely generated the observation.
- Generative models are also popular in machine learning since they can be used to learn structure in unlabelled data (unsupervised learning).
- Much (most) of the information in the world is unlabelled and unsupervised methods that can learn structure are key to progress in AI.
- Generative models also enable us to construct ('phantasise/hallucinate') data, with needing an input. This is done by sampling h from $p(h)$ and then v from $p(v|h)$.

Variational Inference

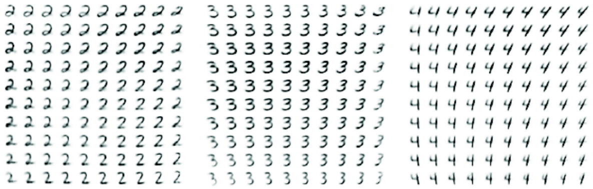
Consider a distribution

$$p(v|\theta) = \int_h p(v|h,\theta)p(h)$$

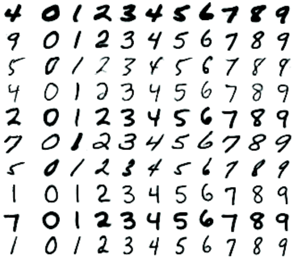
and that we wish to learn θ to maximise the probability this model generates observed data.

$$\log p(v|\theta) \geq -\int q(h|v,\phi) \log q(h|v,\phi) + \int_h q(h|v,\phi)p(v|h,\theta) + \text{const.}$$

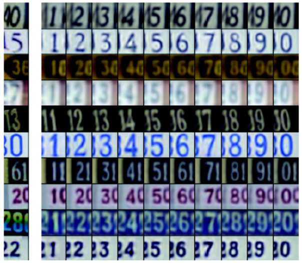
- Idea is to choose a 'variational' distribution $q(h|v,\phi)$ such that we can either calculate analytically the bound, or sample it efficiently.
- We then jointly maximise the bound wrt ϕ and θ.
- We can parameterise $p(v|h,\theta)$ using a deep network. Where exact expectation is intractable, sampling can be used.
- Very popular approach – see 'variational autoencoder' (a misnomer since this is not a variational form of an autoencoder – it's a generative model).
- Extension to semi-supervised method using $p(v) = \int_h \sum_c p(v|h,c)p(c)p(h)$



(a) Handwriting styles for MNIST obtained by fixing the class label and varying the 2D latent variable z



(b) MNIST analogies



(c) SVHN analogies

Figure 1: **(a)** Visualisation of handwriting styles learned by the model with 2D z-space. **(b,c)** Analogical reasoning with generative semi-supervised models using a high-dimensional z-space. The leftmost columns show images from the test set. The other columns show analogical fantasies of x by the generative model, where the latent variable z of each row is set to the value inferred from the test-set image on the left by the inference network. Each column corresponds to a class label y.

DRAW

Discussion

Training Nets

- The surface $E(\mathcal{W})$ is typically highly complex 🔴
- Apart from permutation symmetries (which cause local optima) there are typically many non-trivial local optima 🔴
- There are also typically large parts of the space in which the E surface is almost flat 🔴
- There is little real understanding about how to best train NNs 🔴
- The quality of the solution is often highly dependent on the initialisation of the weights 🔴
- The field is replete with initialisation and regularisation heuristics 🔴
- When well trained, these techniques provide state-of-the-art performance 🟢!

Summary

Neural nets (deep learning) are complex hierarchical functions that are primarily motivated by analogies to information processing in natural systems.

- They are potentially very powerful function approximators 🟢
- Once trained, they are very fast to use 🟢
- They can be scaled up to very large datasets 🟢
- Training is complex – the objective function contains many local optima 🔴
- Parameter initialisation is difficult and critical, especially in networks with many layers 🔴
- Many things that can be adjusted – number of nodes in each layer, number of layers, types of transfer functions, regularisation types, penalty strengths 🔴
- Currently the state-of-the-art in several areas (object recognition, natural language processing, speech recognition) 🔴
- Super hot topic at the moment, with intense interest from big companies 🟢