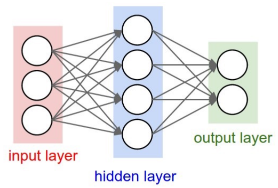
Review

Multi-level networks

- A single artificial neuron has limited computing power
  - Interesting networks have at least three* layers



  - We will restrict our attention to feed-forward networks (no arrows from later stages going back to earlier ones)

*Some (e.g., Bishop) count the number of arrow blocks, so this would be a two layer network.

1

Review

Layering and composition

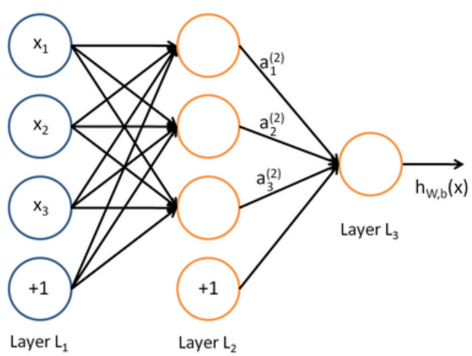
- Our three layer network can be expressed as the composition $\mathbf{y} = f(\mathbf{x}) = f_2(f_1(\mathbf{x}))$

- Similarly "deep" networks are bigger compositions

- Note that we like the functions to be differentiable, and the chain rule is going to be useful.

2

Review



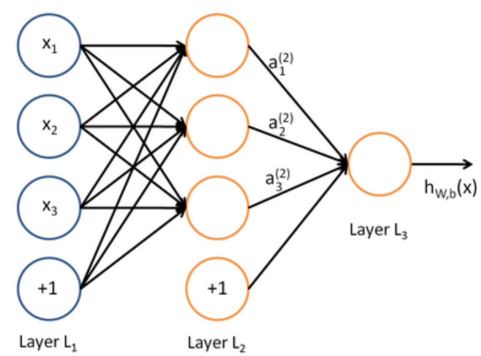
$a_i^{(1)} = x_i$
(input layer)

For $l \geq 1$ $\begin{cases} z_i^{(l+1)} = \sum_{j=1}^{N^{(l)}} W_{ij}^{(l)} a_i^{(l)} + b_i^{(l)} \\ \text{followed by} \quad a_i^{(l+1)} = f\left(z_i^{(l+1)}\right) \end{cases}$

$$f(z) = \frac{1}{1+\exp(-z)}$$ (for example)

3

Review



$$\mathbf{z}^{(l+1)} = W^{(l)}\mathbf{a}^{(l)} + \mathbf{b}^{(l)}$$
$$\mathbf{a}^{(l+1)} = f\left(\mathbf{z}^{(l+1)}\right)$$

(Matrix-vector form,
f(•) is applied element-wise)

4

Review

Training

- Tweak the weights so that for each training instance, the output (e.g., label) for each input vector (e.g., image) is close to correct

Review

Training

- Tweak the weights so that for each training instance, the output (e.g., label) for each input vector (e.g., image) is close to correct
- Optimize the objective function:

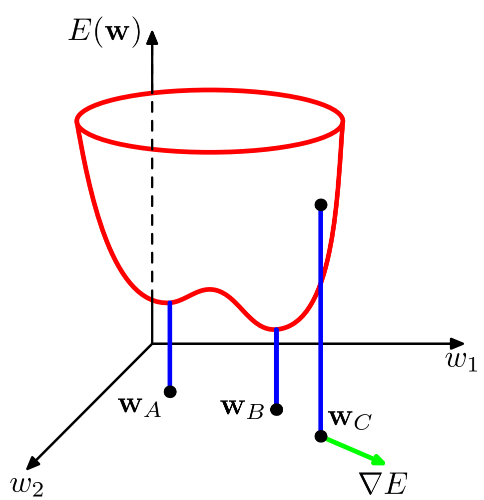
$$J(W,b)=\left[\frac{1}{m}\sum_{i=1}^{m}\left(\frac{1}{2}\left\|h_{W,b}\left(\mathbf{x}^{(i)}\right)-\mathbf{y}^{(i)}\right\|^{2}\right)\right]+\frac{\lambda}{2}\sum_{l=1}^{L-1}\sum_{i=1}^{S_l}\sum_{j=1}^{S_{l+1}}\left(W_{ji}^{(l)}\right)^{2}$$

Training error, superscripts are training points

Neural networks are prone to overfitting, so regularize. This regularization term is just one example—there are many others.

Review

Training



Training

- We consider computing the gradient, and what we do with it as separate tasks
- We generally compute the gradient in training
- We might compute it due to a specific data point (on-line) or for all points considered together
- Once we estimate the gradient we then might
  - Follow it (e.g., conjugate gradient descent)
  - Follow it stochastically
  - Pass it to some other fancy optimizer

Computing the gradient

- A naive way is to simply compute finite differences for each parameter (weight):

$$\frac{\partial E}{\partial W_{i,j}^{(l)}} \cong \frac{E\left(\mathbf{w}^{new}\right) - E\left(\mathbf{w}\right)}{\Delta}$$

where $\mathbf{w}^{new}$ uses $W_{i,j}^{(l)} + \Delta$ instead of $W_{i,j}^{(l)}$

- What is the complexity for each training step?

Computing the gradient

- A naive way is to simply compute finite differences for each parameter (weight):

$$\frac{\partial E}{\partial W_{i,j}^{(l)}} \cong \frac{E\left(\mathbf{w}^{new}\right) - E\left(\mathbf{w}\right)}{\Delta}$$

where $\mathbf{w}^{new}$ uses $W_{i,j}^{(l)} + \Delta$ instead of $W_{i,j}^{(l)}$

- Complexity for each training step is $O\left(W^2\right)$

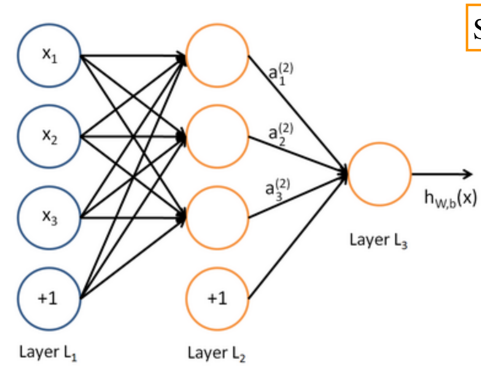
Details skipped

Back propagation

- Better way is to use the chain rule
- First compute a forward pass of the network, and keep track of all intermediate activations
- Now we work backwards level by level
- We will start by looking at the calculus of weights and nodes
- We will look at the gradient due to one training point
  - You can simply add the effect of multiple points

Slide skipped

Recall



$a_i^{(1)} = x_i$
(input layer)

For $l \geq 1$ $\begin{cases} z_i^{(l+1)} = \sum_{j=1}^{N^{(l)}} W_{ij}^{(l)} a_i^{(l)} + b_i^{(l)} \\ \text{followed by} \quad a_i^{(l+1)} = f\left(z_i^{(l+1)}\right) \end{cases}$

(In what follows we include the bias in W)

Slide skipped

$$\delta_j^{(l)} = \frac{\partial E}{\partial z_j^{(l)}} = \sum_k \frac{\partial E}{\partial z_k^{(l+1)}} \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}}$$

$$= \sum_k \frac{\partial E}{\partial z_k^{(l+1)}} f'\left(z_j^{(l)}\right) W_{k,j}^{(l)}$$

$$= f'\left(z_j^{(l)}\right) \sum_k \delta_k^{(l+1)} W_{k,j}^{(l+1)}$$

Slide skipped

$z_j^{(l)}$ only effects E through connections to the next layer

So, $$\frac{\partial E}{\partial z_j^{(l)}} = \frac{\partial E\left(z_1^{(l+1)}, z_2^{(l+1)}, z_3^{(l+1)}, \ldots\right)}{\partial z_j^{(l)}}$$

$$\delta_j^{(l)} = \frac{\partial E}{\partial z_j^{(l)}} = \sum_k \frac{\partial E}{\partial z_k^{(l+1)}} \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}}$$

$$= \sum_k \frac{\partial E}{\partial z_k^{(l+1)}} f'\left(z_j^{(l)}\right) W_{k,j}^{(l)}$$

$$= f'\left(z_j^{(l)}\right) \sum_k \delta_k^{(l+1)} W_{k,j}^{(l+1)}$$

Slide skipped

For the output layer, O, we only depend on the final activation

$$\delta_j^{(o)} = \frac{\partial E}{\partial z_j^{(o)}} = \frac{\partial J(W,b)}{\partial z_j^{(o)}} = \frac{\partial \frac{1}{2}\left\|a_j^{(o)} - y_j\right\|^2}{\partial z_j^{(o)}} = \left(a_j^{(o)} - y_j\right) f'\left(z_j^{(o)}\right)$$

Slide skipped

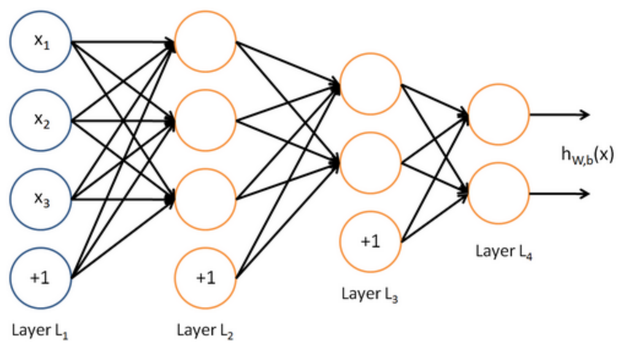
Back propagation algorithm (backprop)

- Do a forward run with data vector, $\mathbf{x}_n$
- Compute the initial $\delta_j^{(o)}$ for each output layer value vs the training data truth using $\delta_j^{(o)} = \left(a_j^{(o)} - y_j\right) f'\left(z_j^{(o)}\right)$
- Back propagate the δ's using $\delta_j^{(l)} = f'\left(z_j^{(l)}\right) \sum_k \delta_k^{(l+1)} W_{k,j}^{(l)}$
- Compute gradient components using $\frac{\partial E}{\partial W_{i,j}^{(l)}} = \delta_j^{(l+1)} a_i^{(l)}$

Different Architectures

- Differences in layers, connectivity, activation functions

"densely" or fully connected, 2 output, *feedforward* network



Activation Functions

Sigmoid

$$f(z) = \frac{1}{1+\exp(-z)}$$

$$f'(z) = f(z)(1-f(z))$$

Tanh

$$f(z) = \tanh(z) = \frac{\exp(z)-\exp(-z)}{\exp(z)+\exp(-z)}$$

$$f'(z) = 1 - \tanh^2(z)$$

Rectified Linear Unit (ReLU)

$$f(z) = \max(0, z)$$

$$f'(z) = \begin{cases} 0 & \text{if } z \le 0 \\ 1 & \text{if } z > 0 \end{cases}$$

Leaky ReLU

$$f(z) = \begin{cases} \alpha z & \text{if } z < 0 \\ z & \text{if } z \ge 0 \end{cases}$$

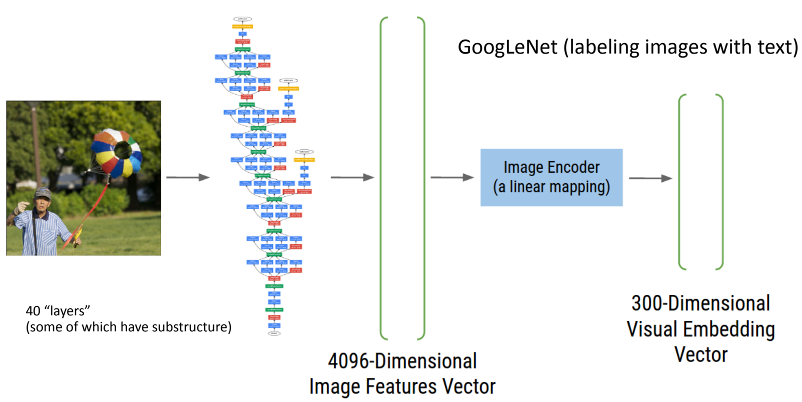
$$f'(z) = \begin{cases} \alpha & \text{if } z \le 0 \\ 1 & \text{if } z > 0 \end{cases}$$

Maxout

$$\max(w_1^\top x + b_1, w_2^\top x + b_2)$$

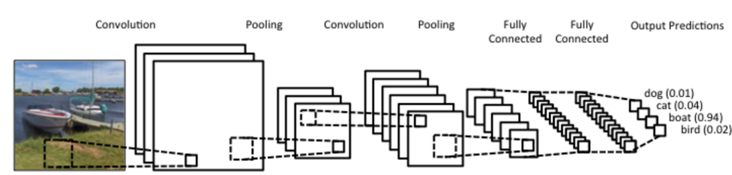
Deep neural nets

Pretty much anything > 3 layers



Convolutional neural network

- Convolutional neural networks tie weights so that the linear part implements convolution

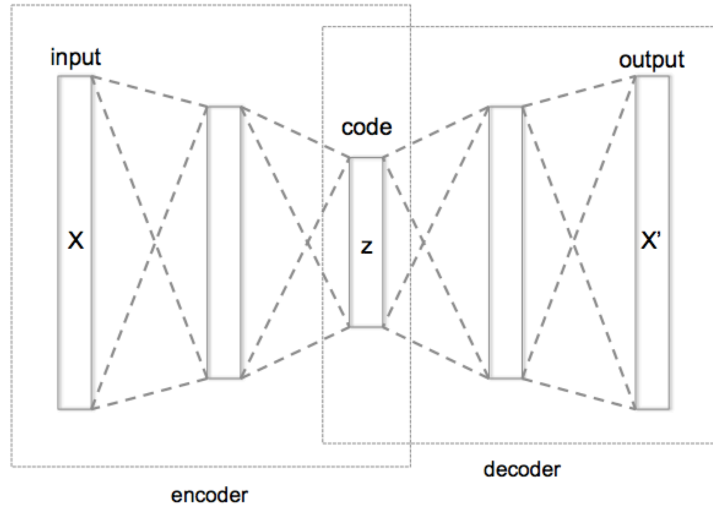


- In a typical *convnet* successive layers become smaller by pooling, and representation (ideally) gets less localized and more semantic

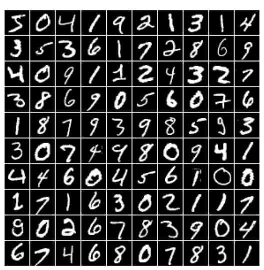
Autoencoder

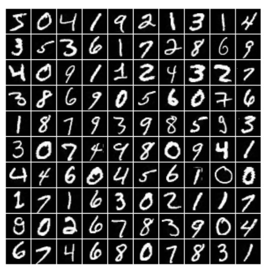
- An autoencoder maps inputs to copies of themselves
- This mostly only makes sense if the intermediate layers bottleneck down to a sparser representation, usually implying fewer (active) neurons
- The idea is to discover a lower dimensional representation that can approximately reconstruct the data
  – Note similarity with PCA
  – Note the obvious relation to compression

Autoencoder



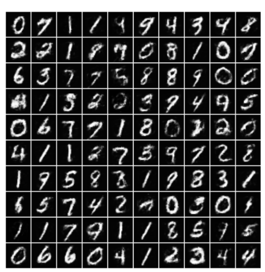
Training Input





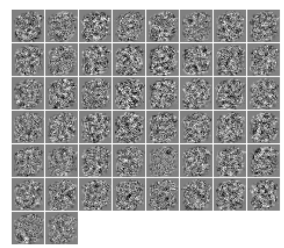
Inferred output after training

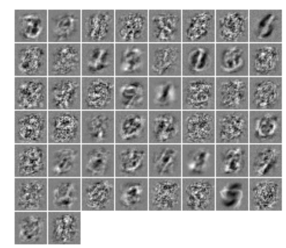
Previously unseen input



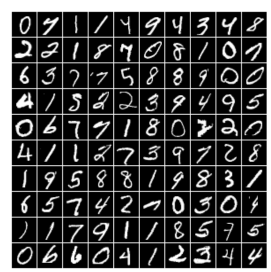
Reconstruction of previously unseen images

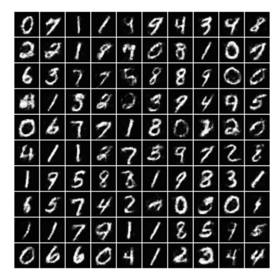
Layer 1 weights for autoencoder with just weight regularization



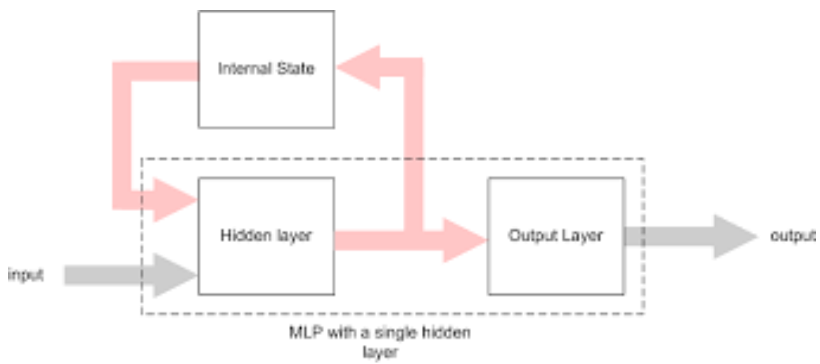


Layer 1 weights for autoencoder with just weight regularization Plus hidden layer sparsity constraint. Note the structure





Recurrent neural network (RNN)



Trained, e.g., with Backpropagation Through Time (BPTT)

(Long-short term memory) LSTM NN