ISTA 421 + INFO 521
Introduction to Machine Learning

Lecture 25:
Neural Networks III – The Sparse Autoencoder

Clay Morrison
claytonm@email.arizona.edu
Harvill 437A
Phone 621-6609

27 November 2017

1

Rest of the Semester

- Mon, Nov 20 – Intro to NN
- Wed, Nov 22 – NN 2: Backpropagation
- Mon, Nov 27 – NN 3: Sparse Autoencoders
- Wed, Nov 29 – Clustering: K-Means
- Mon, Dec 4 – Clustering: Gaussian Mixture Model
- Wed, Dec 6 – Principle Components Analysis

- Homework 5 (NN) due Thurs, Dec 7
- Final Project due Mon, Dec 11

- **Reminder: Please fill out course evaluation (TCE)**

2

Neural Networks and Deep Learning

Gradient Descent

- Given a loss function

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\frac{1}{2}\left(\underbrace{\|h_\theta(x^{(i)}) - y^{(i)}\|}_{\substack{L_2 \text{ norm (i.e., Euclidean distance)} \\ \text{when } h_\theta \text{ and } y \text{ are vectors; when scalars, just subtract}}}\right)^2$$

- *m* is the number of examples and *h* some function of parameters θ
- Gradient descent updates the parameters in steps:

$$\theta_j = \theta_j - \alpha\frac{\partial J(\theta)}{\partial \theta_j}$$

Backpropagation Algorithm
(for fully connected feedforward network)

training pairs: $\{(x^{(1)}, y^{(1)}), (x^{(m)}, y^{(m)})\}$

Define the loss

$$J(W,b) = \left[\frac{1}{m}\sum_{i=1}^{m}\left(\frac{1}{2}\underbrace{\|h_{W,b}(x^{(i)}) - y^{(i)}\|^2}_{\text{Implementation note: use } \texttt{numpy.linalg.norm()}}\right)\right] + \underbrace{\text{weight decay term}}_{\text{(note that reg term typically not over bias terms)}}$$

Neural networks can be prone to overfitting, so regularize

Goal is to minimize $J(W,b)$

One iteration of gradient descent updates parameters W, b as follows:

$$W_{ij}^{(l)} = W_{ij}^{(l)} - \alpha\frac{\partial J(W,b)}{\partial W_{ij}^{(l)}} \qquad \frac{\partial J(W,b)}{\partial W_{ij}^{(l)}} = \left[\frac{1}{m}\sum_{k=1}^{m}\frac{\partial J(W,b;x^{(k)},y^{(k)})}{\partial W_{ij}^{(l)}}\right] + \lambda W_{ij}^{(l)}$$

$$b_i^{(l)} = b_i^{(l)} - \alpha\frac{\partial J(W,b)}{\partial b_i^{(l)}} \qquad \frac{\partial J(W,b)}{\partial b_i^{(l)}} = \frac{1}{m}\sum_{k=1}^{m}\frac{\partial J(W,b;x^{(k)},y^{(k)})}{\partial b_i^{(l)}}$$

SISTA 5

Backpropagation Algorithm
(for fully connected feedforward network)

Partial derivatives of cost function for single example (x,y): $\frac{\partial J(W,b;x,y)}{\partial W_{ij}^{(l)}}$ $\frac{\partial J(W,b;x,y)}{\partial b_i^{(l)}}$

Backprop Core (Vectorized):

'•' denotes element-wise product (Hadamard product)
$f'([z_1,z_2,z_3]) = [f'(z_1), f'(z_2), f'(z_3)]$

(1) Perform feedforward pass

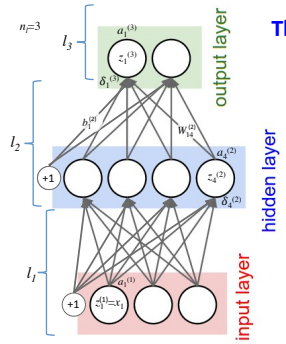
(2) For the output layer (layer n_l), set $\delta^{(n_l)} = -(y - a^{(n_l)}) \bullet f'(z^{(n_l)})$

(3) For $l = n_l - 1, n_l - 2, n_l - 3, \ldots, 2$, set $\delta^{(l)} = \left((W^{(l)})^\top \delta^{(l+1)}\right) \bullet f'(z^{(l)})$

(4) Compute the desired partial derivatives, which are given as:

$$\nabla_{W^{(l)}} J(W,b;x,y) = \delta^{(l+1)}(a^{(l)})^\top$$
$$\nabla_{b^{(l)}} J(W,b;x,y) = \delta^{(l+1)}$$

If sigmoid activation: $f'(z_i^{(l)}) = a_i^{(l)}(1 - a_i^{(l)})$ (store during forward pass)



The Complete Backprop Algorithm as Gradient Descent:

(1) Set $\Delta W^{(l)} := 0,\ \Delta b^{(l)} := 0$ for all l (note that 'ΔW' and 'Δb' are single variables)

(2) For i = 1 to m,

(a) Use backpropagation (Backprop Core) to compute $\nabla_{W^{(l)}} J(W,b;x,y)$ $\nabla_{b^{(l)}} J(W,b;x,y)$

(b) Set $\Delta W^{(l)} := \Delta W^{(l)} + \nabla_{W^{(l)}} J(W,b;x,y)$

(c) Set $\Delta b^{(l)} := \Delta b^{(l)} + \nabla_{b^{(l)}} J(W,b;x,y)$

(3) Update the parameters:

$$W^{(l)} = W^{(l)} - \alpha\underbrace{\left[\left(\frac{1}{m}\Delta W^{(l)}\right) + \lambda W^{(l)}\right]}_{\text{gradient}} \qquad b^{(l)} = b^{(l)} - \alpha\underbrace{\left[\frac{1}{m}\Delta b^{(l)}\right]}_{\text{gradient}}$$

Repeat (2) and (3) until "convergence"

Initializing Parameters

- If all parameters start off at identical values (e.g., 0), then all hidden layer units would learn the same function of the input.
- Random initialization breaks symmetry.
- Often sample small random values near zero:

$$\mathcal{N}(0, \epsilon^2) \quad (\text{e.g., } \epsilon = 0.01)$$

- Another heuristic:

$$Uniform\left[-\sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}} + 1}}, \sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}} + 1}}\right]$$

Improvements to Gradient Descent

$$W^{(l)} = W^{(l)} - \alpha\left[\left(\frac{1}{m}\Delta W^{(l)}\right) + \lambda W^{(l)}\right] \qquad b^{(l)} = b^{(l)} - \alpha\left[\frac{1}{m}\Delta b^{(l)}\right]$$

- Automatically adapting α
- Newton's method(s): using Hessian information
- L-BFGS
- Conjugate Gradient

What you need to provide:

A function that returns the cost ($J(\theta)$) and grad ($\nabla_\theta J(\theta)$)

Initial theta

```
J = lambda x: your_fn(x, vis_size, hid_size, lambda_, data)
scipy.optimize.minimize(fun=J, x0=theta, method='L-BFGS-B')
```

Debugging: Gradient Checking

Numerically checking derivatives

Recall mathematical definition of the derivative (assuming J is fn of θ):

$$\frac{\partial J(\theta)}{\partial \theta} = \lim_{\epsilon \to 0} \frac{J(\theta + \epsilon) - J(\theta - \epsilon)}{2\epsilon}$$

At any specific value of θ we can numerically approximate the derivative by:

$$\frac{J(\theta + \text{EPSILON}) - J(\theta - \text{EPSILON})}{2 \times \text{EPSILON}}$$

In practice, set EPSILON to a small constant, such as $10^{-4} = 0.0001$.

You can use this numerical approximation to compare against a function $g(\theta) = \frac{\partial J(\theta)}{\partial \theta}$ used to compute the gradient.

$$g(\theta) \approx \frac{J(\theta + \text{EPSILON}) - J(\theta - \text{EPSILON})}{2 \times \text{EPSILON}}$$

With EPSILON = 10^{-4}, you'll usually get a similarity between g(θ) and your numerical approximation to at least 4 significant digits.

Debugging: Gradient Checking

Numerically checking derivatives $g(\theta) \approx \frac{J(\theta + \text{EPSILON}) - J(\theta - \text{EPSILON})}{2 \times \text{EPSILON}}$ $g(\theta) = \frac{\partial J(\theta)}{\partial \theta}$

The above approximates the gradient for a single parameter θ.
If we are computing the derivative of each component parameter i in vector θ, then we can modify the above estimate by varying the specific parameter value by EPSILON:

$$g_i(\boldsymbol{\theta}) = \frac{\partial J(\boldsymbol{\theta})}{\partial \theta_i} \qquad \boldsymbol{\theta} = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_i \\ \vdots \\ \theta_N \end{bmatrix} \qquad \mathbf{e}_i = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix}$$

Let $\boldsymbol{\theta}^{(i+)} = \boldsymbol{\theta} + \text{EPSILON} \times \mathbf{e}_i$
$\boldsymbol{\theta}^{(i-)} = \boldsymbol{\theta} - \text{EPSILON} \times \mathbf{e}_i$

The comparison then becomes:

$$g_i(\boldsymbol{\theta}) \approx \frac{J(\boldsymbol{\theta}^{(i+)}) - J(\boldsymbol{\theta}^{(i-)})}{2 \times \text{EPSILON}}$$

The goal of each iteration of the Backpropagation Algorithm is to compute the gradient

$$W^{(l)} = W^{(l)} - \alpha \underbrace{\left[\left(\frac{1}{m} \Delta W^{(l)} \right) + \lambda W^{(l)} \right]}_{\text{gradient}} \qquad \nabla_{W^{(l)}} J(W, b)$$

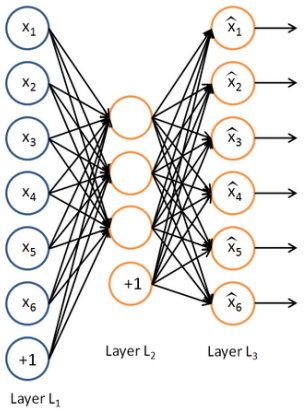
$$b^{(l)} = b^{(l)} - \alpha \underbrace{\left[\frac{1}{m} \Delta b^{(l)} \right]}_{\text{gradient}} \qquad \nabla_{b^{(l)}} J(W, b)$$

The above numerical gradient estimation can be used to numerically compute the derivatives of $J(W, b)$ and compare to the computations of $\nabla_{W^{(l)}} J(W, b)$, $\nabla_{b^{(l)}} J(W, b)$

The Autoencoder

An **autoencoder** is a neural network architecture supporting unsupervised learning, using backpropagation in a setting where the target output values are **equal** to the inputs:

$$\{x^{(1)}, x^{(2)}, x^{(3)}, ...\}, x^{(i)} \in \mathbb{R}^n \qquad y^{(i)} = x^{(i)}$$



The autoencoder tries to learn a function $h_{W,b}(x) \approx x$

That is, it is trying to learn an approximation to the **identity function**.

With fewer hidden units, it has to learn a **compressed representation** of the input.

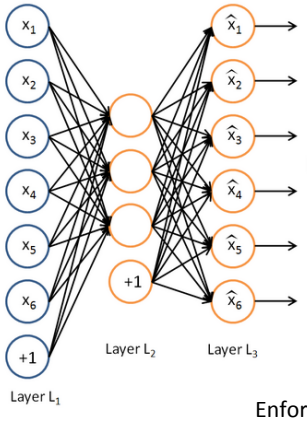
Given hidden unit activations, it attempts to **reconstruct** the input pattern (at the output layer).

If there is structure in the data (e.g., if some of the input features are correlated), then the autoencoder may discover some of those correlations.
This amounts to learning a lower-dimensional representation of the input.

The Autoencoder

An **autoencoder** is a neural network architecture supporting unsupervised learning, using backpropagation in a setting where the target output values are **equal** to the inputs:

$$\{x^{(1)}, x^{(2)}, x^{(3)}, ...\}, x^{(i)} \in \mathbb{R}^n \qquad y^{(i)} = x^{(i)}$$



Even when the number of hidden units is large (possibly larger than the number of inputs) we can still discover interesting structure by imposing a **sparsity constraint** on the hidden units.

Informally, think of a hidden unit as "active" ("firing) when activation is close to 1, and "inactive" when close to 0.

Let $a_j^{(2)}(x)$ denote the activation of hidden unit when the network is given specific input x.

Let $\hat{\rho}_j = \frac{1}{m} \sum_{i=1}^{m} \left[a_j^{(2)}(x^{(i)}) \right]$ … i.e., the average activation of the hidden unit j (averaged over the training set)

Enforce the constraint: $\hat{\rho}_j = \rho$

Sparsity parameter

Enforcing Sparsity

$$\hat{\rho}_j = \frac{1}{m}\sum_{i=1}^{m}\left[a_j^{(2)}(x^{(i)})\right]$$ Enforce the constraint: $\hat{\rho}_j = \rho$ **Sparsity parameter**

Add extra penalty term to the optimization objective that penalizes $\hat{\rho}_j$ for deviating significantly from ρ

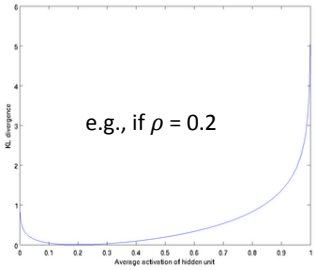
We will use the following: $\sum_{j=1}^{s_2} \overbrace{\rho \log \frac{\rho}{\hat{\rho}_j} + (1-\rho)\log\frac{1-\rho}{1-\hat{\rho}_j}}^{D_{\mathrm{KL}}(\rho\|\hat{\rho}_j)}$

s_2 is the number of neurons in the hidden layer, index *j* is summing over the hidden units in the network.

This is a version of the **Kullback-Leibler (KL) divergence**. In this case, it's the KL divergence between a Bernoulli random variable with mean ρ and a Bernoulli random variable with mean $\hat{\rho}_j$

$$\sum_{j=1}^{s_2} D_{\mathrm{KL}}(\rho\|\hat{\rho}_j)$$

$$D_{\mathrm{KL}}(P\|Q) = \sum_x P(x)\log\frac{P(x)}{Q(x)}$$



e.g., if $\rho = 0.2$

Recall: $J(W,b) = \left[\frac{1}{m}\sum_{i=1}^{m}\left(\frac{1}{2}\|h_{W,b}(x^{(i)}) - y^{(i)}\|^2\right)\right] + \frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}\left(W_{ji}^{(l)}\right)^2$ Recall: $\delta_i^{(l)} = \left(\sum_{j=1}^{s_{l+1}} W_{ji}^{(l)}\delta_j^{(l+1)}\right) f'(z_i^{(l)})$

Enforcing Sparsity

$$\hat{\rho}_j = \frac{1}{m}\sum_{i=1}^{m}\left[a_j^{(2)}(x^{(i)})\right]$$ Enforce the constraint: $\hat{\rho}_j = \rho$ **Sparsity parameter**

Must be computed across all data (at feedforward)

Add extra penalty term to the optimization objective that penalizes $\hat{\rho}_j$ for deviating significantly from ρ

We will use the following: $\sum_{j=1}^{s_2} \overbrace{\rho \log \frac{\rho}{\hat{\rho}_j} + (1-\rho)\log\frac{1-\rho}{1-\hat{\rho}_j}}^{D_{\mathrm{KL}}(\rho\|\hat{\rho}_j)}$

s_2 is the number of neurons in the hidden layer, index j is summing over the hidden units in the network.

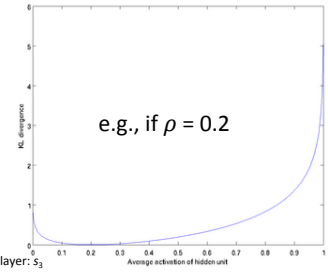
The overall cost function is:

$$J_{\mathrm{sparse}}(W,b) = J(W,b) + \beta\sum_{j=1}^{s_2} D_{\mathrm{KL}}(\rho\|\hat{\rho}_j)$$

For the derivative, simply replace $\delta_i^{(2)}$ with:

$$\delta_i^{(2)} = \left(\left(\sum_{j=1}^{s_3} W_{ji}^{(2)}\delta_j^{(3)}\right) + \beta\left(-\frac{\rho}{\hat{\rho}_i} + \frac{1-\rho}{1-\hat{\rho}_i}\right)\right) f'(z_i^{(2)})$$

Note: the UFLDL tutorial get's this index wrong: they say s_2 but it should be the number of nodes in *next* layer: s_3

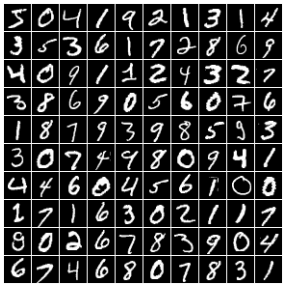


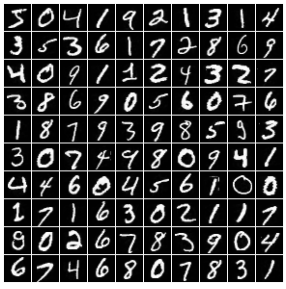
e.g., if $\rho = 0.2$

The Task: MNIST

28 x 28 images (784 pixels)

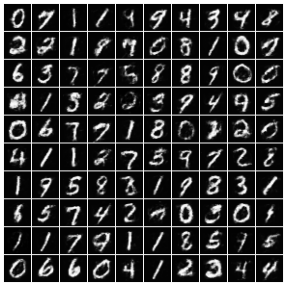
Training Input





Inferred output after training

Previously unseen input

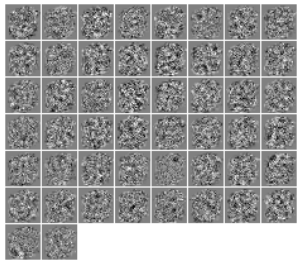


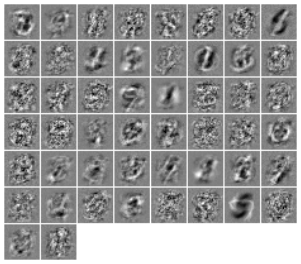
Reconstruction of previously unseen images

The Task: MNIST

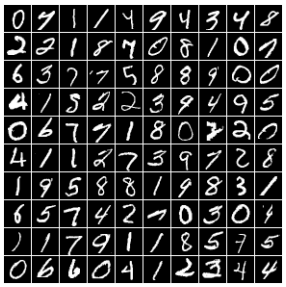
28 x 28 images (784 pixels)

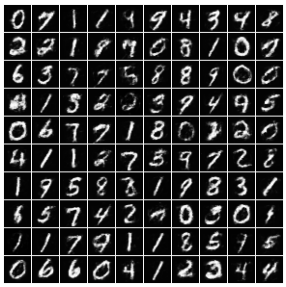
Layer 1 weights for autoencoder with just weight regularization





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





Softmax Regression

Suppose we want to classify

Recall logistic regression, where we compute the probability of belonging to class 1.
Here we can express this output in terms of the output activation:

$$P(y=1|x,\theta) = \frac{1}{1+\exp(-\theta^\top x)} = h_\theta(x) \qquad P(y=0|x,\theta) = \frac{\exp(-\theta^\top x)}{1+\exp(-\theta^\top x)}$$

And 1 minus this would be the probability of class zero.
When training logistic reg, we then combined these two probabilities to define the cost:

$$J(\theta) = -\left[\sum_{i=1}^{m} y^{(i)} \log h_\theta(x^{(i)}) + (1-y^{(i)}) \log(1-h_\theta(x^{(i)}))\right]$$

But how do we represent more than two classes? $y^{(i)} \in \{1, 2, ..., K\}$

$$h_\theta = \begin{bmatrix} P(y=1|x;\theta) \\ P(y=2|x;\theta) \\ \vdots \\ P(y=K|x;\theta) \end{bmatrix} = \frac{1}{\sum_{j=1}^{K} \exp(\theta^{(j)\top} x)} \begin{bmatrix} \exp(-\theta^{(1)\top} x) \\ \exp(-\theta^{(2)\top} x) \\ \vdots \\ \exp(-\theta^{(K)\top} x) \end{bmatrix}$$

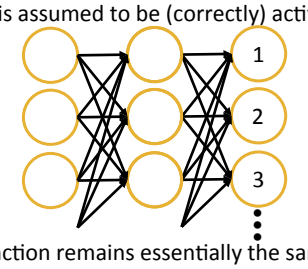
And the cost function generalizes to:

$$J(\theta) = -\left[\sum_{i=1}^{m}\sum_{k=1}^{K} \mathbb{1}_{\{y^{(i)}=k\}} \log \frac{\exp(-\theta^{(k)\top} x^{(i)})}{\sum_{j=1}^{K} \exp(-\theta^{(j)\top} x^{(i)})}\right]$$

Softmax Regression

Suppose we want to classify

Softmax regression is a single layer. It can be added to the output layer of any multi-layer perceptron, representing an output where each output layer node represents a class, and only one is assumed to be (correctly) active at a given time.



The softmax cost function remains essentially the same, but now compares the output layer activations to the target output

$$J(\theta) = -\left[\sum_{i=1}^{m}\sum_{k=1}^{K} \mathbb{1}_{\{y^{(i)}=k\}} \log \frac{\exp(\theta^{(k)\top} h_{W,b}(x^{(i)}))}{\sum_{j=1}^{K} \exp(\theta^{(j)\top} h_{W,b}(x^{(i)}))}\right]$$

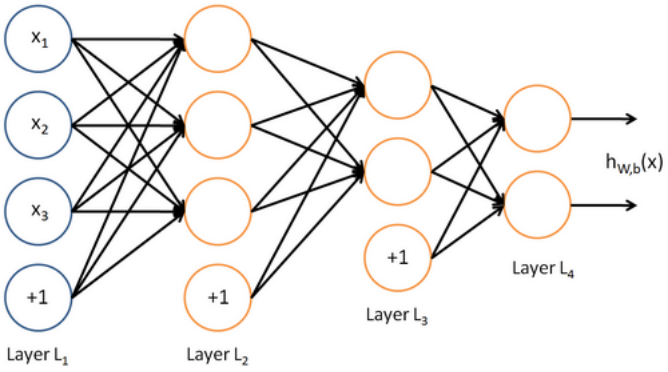
And only the $\delta^{(n_l)}$ term for the output layer is changed – the rest of the delta computations for all of the other layers remains the same as in backprop.

$$\delta^{(n_l)} = -\sum_{i=1}^{m}\left[\mathbb{1}_{\{y^{(i)}=k\}} - P(y^{(i)}=k|x^{(i)};\theta)\right]$$

Different Architectures

- Differences in layers, connectivity

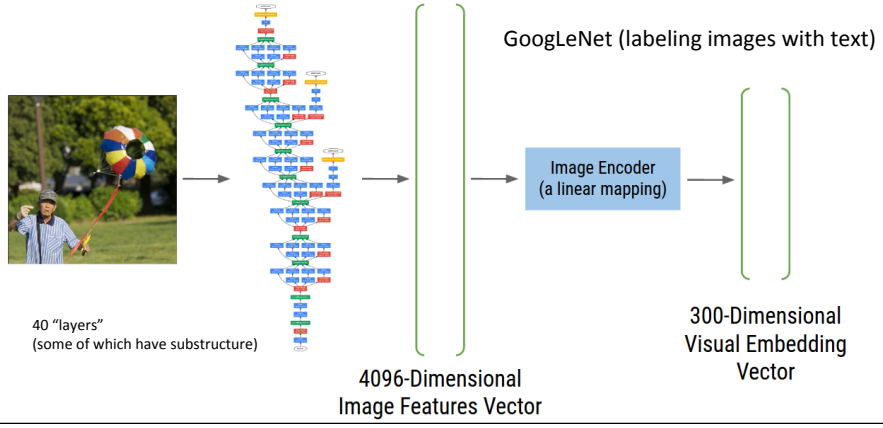
"densely" or fully connected, 2 output, *feedforward* network



Different Architectures

- Differences in layers, connectivity

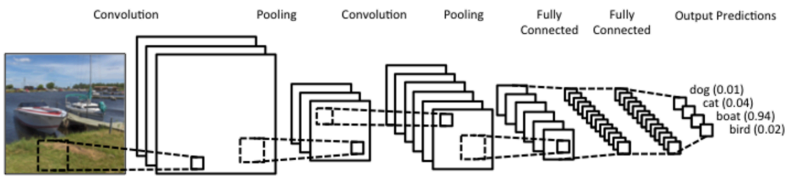
"Deep" neural nets. Pretty much anything > 3 layers



Different Architectures

- Differences in layers, connectivity

Convolutional neural network

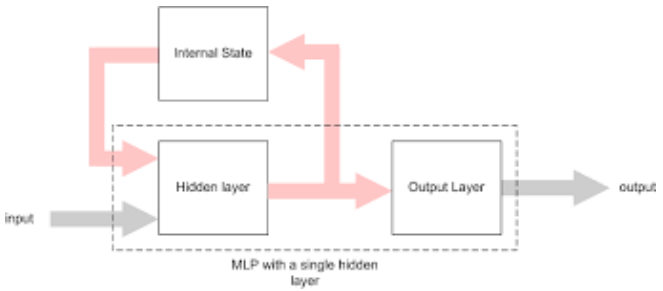


https://www.youtube.com/watch?v=qrzQ_AB1DZk

Different Architectures

- Differences in layers, connectivity

Recurrent neural network



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

Different Architectures

- Differences in layers, connectivity

Long Short Term Memory (LSTM) neural network

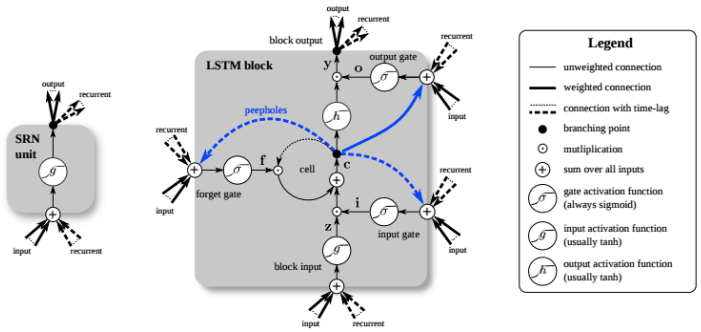


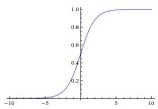
Figure 1. Detailed schematic of the Simple Recurrent Network (SRN) unit (left) and a Long Short-Term Memory block (right) as used in the hidden layers of a recurrent neural network.

Activation Function Redux

Sigmoid

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

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

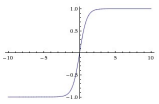


Historically popular: like activation of neuron (0 = saturated, 1 = high firing rate)
Rarely used now; drawbacks:
(1) **Vanishing gradient problem**: saturates in 0 or 1 regions to nearly zero gradient (hence, sensitive to initialization, and during learning).
(2) **Outputs are not zero-centered**: leads to instability during gradient descent (if incoming data to neuron is always positive, then gradient on weights during backprop will always be all positive or all negative). Not as bad as issue (1)

Tanh

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

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



A "scaled" sigmoid.

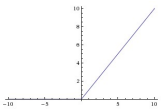
$$\tanh(z) = 2\sigma(2z) - 1$$

Is zero-centered, so universally preferred over sigmoid

Rectified Linear Unit (ReLU)

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

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



Now very popular
(+) Greatly accelerates (e.g., x6) convergence of stochastic gradient descent compared to sigmoid/tanh; believed due to linear, non-saturating form
(+) Compared to expensive sigmoid/tanh operations (exponentials), ReLU can be implemented by simply threshold matrix of activations at zero.
(-) "dying ReLU" problem: gradient update could go below zero and never be activated again. If learning rate too high, leads to "dead" neurons (up to 40%) that never activated across the entire training dataset. Can be minimized with proper learning rate.

Leaky ReLU

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

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

Attempts to fix "dying ReLU" problem.
Instead of 0 when x < 0, have a small negative slope, e.g. 0.01.
Sometimes slope in negative region is a parameter of each neuron (Kaiming He et al., 2015)
Mixed results – works for some.

Maxout

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

Non-linearity applied on the dot product between the weights and the data
Generalizes ReLU (w1,b1=0) and Leaky ReLu.
Gets benefits of ReLU while avoiding dying ReLU.
BUT, doubles the number of parameters for every neuron.

http://cs231n.github.io/neural-networks-1/

Rare to use different activation functions in same network, although no in principle reason not to.

Software

- TensorFlow: https://www.tensorflow.org/
- Theano: http://deeplearning.net/software/theano/
- Keras: https://keras.io/