Class 9: Neural Networks & Deep Learning

MSA 8150: Machine Learning for Analytics

Alireza Aghasi

Institute for Insight, Georgia State University

Introduction

Topics that Will be Covered Today

- Brief overview of linear algebra and matrix notation
- Matrix notation for linear models, especially multi-output models
- Structure of the brain
- Neural network models in matrix form
- Gradient descent technique for minimization
- NN fitting objetive and (stochastic) gradient descent
- Introduction to signal processing and linear filtering
- Equipping NNs with convolutional layers
- Other variants of NNs, Recurrent NNs, Generative Adversarial Networks

Vectors and Matrices

- A vector is often referred to a 1-D array of numbers

$$\boldsymbol{a} = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \end{bmatrix} \in \mathbb{R}^m, \qquad \text{Ex}: \quad \boldsymbol{a} = \begin{bmatrix} 1.2 \\ -3 \\ 2 \end{bmatrix} \in \mathbb{R}^3.$$

- A matrix is often referred to a 2-D array of numbers

$$\boldsymbol{A}_{m\times n} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}, \quad \text{Ex: } \boldsymbol{A} = \begin{bmatrix} 4 & -7 & 5 & 0 \\ -2 & 0 & 11 & 8 \\ 19 & 1 & -3 & 12 \end{bmatrix}.$$

Matrix Notation

m-by-n matrix

$a_{i,j}$ $\quad$ n columns $\quad$ j changes

m rows

i changes

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & \cdots \\ a_{2,1} & a_{2,2} & a_{2,3} & \cdots \\ a_{3,1} & a_{3,2} & a_{3,3} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

More on Matrices

- Transpose of a matrix:

$$\boldsymbol{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \in \mathbb{R}^{m \times n}, \;\; \boldsymbol{A}^\top = \begin{bmatrix} a_{11} & a_{21} & \cdots & a_{m1} \\ a_{12} & a_{22} & \cdots & a_{m2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \cdots & a_{mn} \end{bmatrix} \in \mathbb{R}^{n \times m}$$

- Example:

$$\boldsymbol{A} = \begin{bmatrix} 4 & -7 & 5 & 0 \\ -2 & 0 & 11 & 8 \\ 19 & 1 & -3 & 12 \end{bmatrix}, \;\; \boldsymbol{A}^\top = \begin{bmatrix} 4 & -2 & 19 \\ -7 & 0 & 1 \\ 5 & 11 & -3 \\ 0 & 8 & 12 \end{bmatrix}.$$

Matrix Product

- The product of two matrices $\boldsymbol{A}, \boldsymbol{B}$ with compatible sizes $n \times m$, $m \times p$ is denoted by $\boldsymbol{AB}$ and is of size $n \times p$:

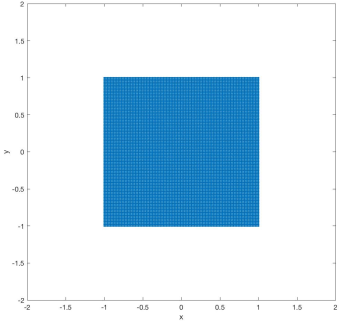
$$\begin{pmatrix} a_{11} & \cdots & a_{1m} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nm} \end{pmatrix} \begin{pmatrix} b_{11} & \cdots & b_{1p} \\ \vdots & \ddots & \vdots \\ b_{m1} & \cdots & b_{mp} \end{pmatrix} = \begin{pmatrix} \sum_{k=1}^{m} a_{1k} b_{k1} & \cdots & \sum_{k=1}^{m} a_{1k} b_{kp} \\ \vdots & \ddots & \vdots \\ \sum_{k=1}^{m} a_{nk} b_{k1} & \cdots & \sum_{k=1}^{m} a_{nk} b_{kp} \end{pmatrix}$$

- Example:

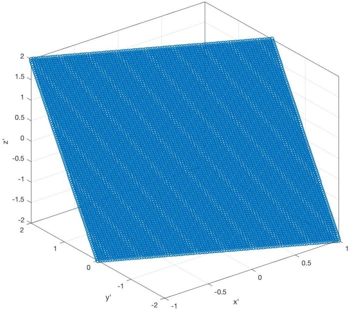
$$\begin{pmatrix} 1 & 0 \\ -1 & 1 \\ 0 & 2 \end{pmatrix} \begin{pmatrix} -2 & 0 \\ 1 & 2 \end{pmatrix} = \begin{pmatrix} -2 & 0 \\ 3 & 2 \\ 2 & 4 \end{pmatrix}$$

Matrices as a Way of Linear Transformation

$$\begin{pmatrix} x_1' & x_2' & \cdots & x_n' \\ y_1' & y_2' & \cdots & y_n' \\ z_1' & z_2' & \cdots & z_n' \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ -1 & 1 \\ 0 & 2 \end{pmatrix} \begin{pmatrix} x_1 & x_2 & \cdots & x_n \\ y_1 & y_2 & \cdots & y_n \end{pmatrix}$$



$\Rightarrow$



Matrix Representation for the Linear Models

- For a linear model we had

$$y = b_0 + w_1 x_1 + \cdots + w_p x_p = w_0 + \boldsymbol{w}^\top \boldsymbol{x},$$

where

$$\boldsymbol{w} = \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_p \end{pmatrix}, \quad \boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_p \end{pmatrix}.$$

- So when we fit the model and want to evaluate it for a number of test points $\boldsymbol{x}^{t_1}, \boldsymbol{x}^{t_2}, \cdots, \boldsymbol{x}^{t_n}$ all we need to do is the following

$$\boldsymbol{y}^t = [y^{t_1}, \cdots, y^{t_n}] = [b_0, b_0, \cdots, b_0] + \boldsymbol{w}^\top \begin{pmatrix} x_1^{t_1} & x_1^{t_2} & \cdots & x_1^{t_n} \\ x_2^{t_1} & x_2^{t_2} & \cdots & x_2^{t_n} \\ \vdots & & & \vdots \\ x_p^{t_1} & x_p^{t_2} & \cdots & x_p^{t_n} \end{pmatrix}$$

Matrix Representation for the Linear Models (Multi Response)

- For a linear model with m responses

$$
\begin{aligned}
y_1 &= b_1 + w_{1,1}x_1 + \cdots + w_{1,p}x_p \\
y_2 &= b_2 + w_{2,1}x_1 + \cdots + w_{2,p}x_p \\
&\vdots \\
y_m &= b_m + w_{m,1}x_1 + \cdots + w_{m,p}x_p
\end{aligned}
$$

which can be written in the matrix form as

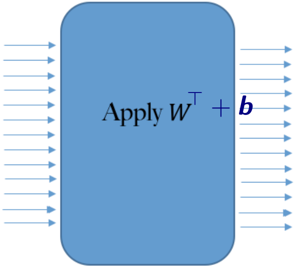
$$\boldsymbol{y} = \boldsymbol{b} + \boldsymbol{W}^\top \boldsymbol{x},$$

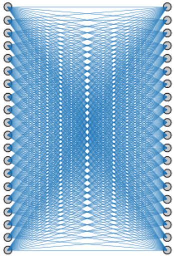
where

$$\boldsymbol{b} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}, \quad \boldsymbol{W} = \begin{pmatrix} w_{1,1} & w_{2,1} & \cdots & w_{m,1} \\ w_{1,2} & w_{2,2} & \cdots & w_{m,2} \\ \vdots & \vdots & \vdots & \vdots \\ w_{1,p} & w_{2,p} & \cdots & w_{m,p} \end{pmatrix}, \quad \boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_p \end{pmatrix}.$$

Fitting Multi Response LMs

- An edge between two nodes is present when $W_{i,j} \neq 0$

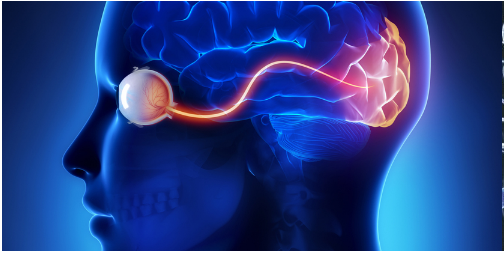


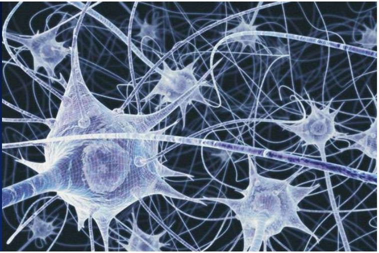


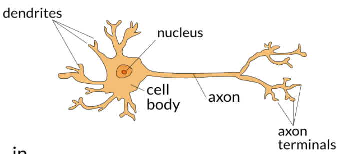
- Suppose we have the training samples $(\boldsymbol{x}_1, \boldsymbol{y}_1), \cdots, (\boldsymbol{x}_N, \boldsymbol{y}_N)$. To fit the model to the training data, we only need to minimize the following RSS:

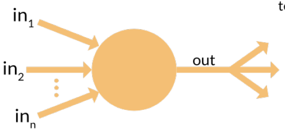
$$\min_{\boldsymbol{W}, \boldsymbol{b}} \sum_{i=1}^{N} \left\| \boldsymbol{y}_i - \boldsymbol{b} - \boldsymbol{W}^\top \boldsymbol{x}_i \right\|^2$$

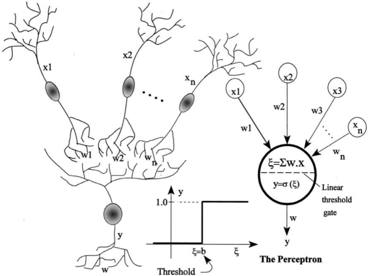
Structure of the Brain











Nonlinear Activation Applied to a Vector

- Sigmoid function:

$$\sigma(x) = \frac{e^x}{1 + e^x}$$

- When sigmoid is applied to a vector or a matrix, it applies to each component individually

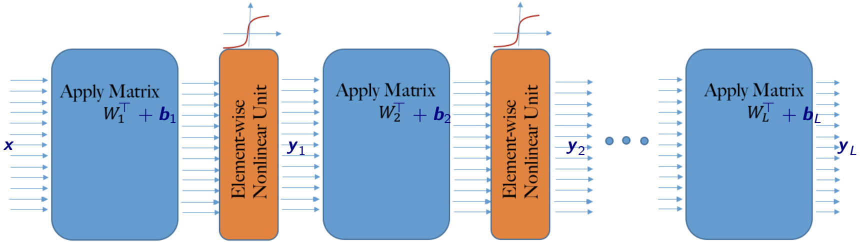
$$\sigma\left(\begin{bmatrix} 3 \\ -2 \\ -1 \end{bmatrix}\right) = \begin{bmatrix} \frac{e^3}{1+e^3} \\ \frac{e^{-2}}{1+e^{-2}} \\ \frac{e^{-1}}{1+e^{-1}} \end{bmatrix}$$

- Another widely used activation is the rectified linear unit:

$$\texttt{ReLU}(x) = \max(x, 0)$$

$$\texttt{ReLU}\left(\begin{bmatrix} 3 \\ -2 \\ -1 \end{bmatrix}\right) = \begin{bmatrix} 3 \\ 0 \\ 0 \end{bmatrix}$$

Standard Architecture of Neural Networks



- A neural network consists of a sequence of multi-output linear units followed by nonlinear activations

$$\boldsymbol{y}_1 = \sigma_1\left(\boldsymbol{W}_1^\top \boldsymbol{x} + \boldsymbol{b}_1\right)$$

$$\boldsymbol{y}_2 = \sigma_2\left(\boldsymbol{W}_2^\top \boldsymbol{y}_1 + \boldsymbol{b}_2\right)$$

$$\vdots$$

$$\boldsymbol{y}_L = \sigma_L\left(\boldsymbol{W}_L^\top \boldsymbol{y}_{L-1} + \boldsymbol{b}_L\right)$$

Standard Architecture of Neural Networks

- Normally all activations are taken to be identical except the last layer
- If we have regression problem, often no activation is used at the output, i.e.,

$$\boldsymbol{y}_L = \boldsymbol{W}_L^\top \boldsymbol{y}_{L-1} + \boldsymbol{b}_L$$

- For classification problems, often a soft-max unit is used at the output, i.e., for $\boldsymbol{y} = [y_1, \cdots, y_m]^\top$

$$\sigma_L(y_i) = \frac{e^{y_i}}{\sum_{j=1}^m e^{y_j}}, \quad i = 1, \cdots, L.$$

- Example:

$$\begin{pmatrix} 0.5 \\ 1.8 \\ -2.3 \\ 0.9 \\ 0.3 \end{pmatrix} \overset{\texttt{soft-max}}{\Rightarrow} \begin{pmatrix} 0.14 \\ 0.52 \\ 0.01 \\ 0.21 \\ 0.12 \end{pmatrix}$$

How to Fit (Train) Neural Networks

- For the proposed architecture, we need to learn $\boldsymbol{W}_1, \cdots, \boldsymbol{W}_L$ and $\boldsymbol{b}_1, \cdots, \boldsymbol{b}_L$
- Let's first derive the function that relates $\boldsymbol{x}$ to $\boldsymbol{y}_L$. Lets define

$$f_\ell(\boldsymbol{z}) = \sigma_\ell\left(\boldsymbol{W}_\ell^\top \boldsymbol{z} + \boldsymbol{b}_\ell\right),$$

then we have

$$\begin{aligned}\boldsymbol{y}_L &= f_L(\boldsymbol{y}_{L-1}) \\ &= f_L(f_{L-1}(\boldsymbol{y}_{L-2})) \\ &\vdots \\ &= f_L(f_{L-1}(f_{L-2}(\cdots f_1(\boldsymbol{x})\cdots)))\end{aligned}$$

- Basically

$$\boldsymbol{y}_L = \mathcal{M}(\boldsymbol{x}), \quad \text{where} \quad \mathcal{M}(\boldsymbol{x}) = f_L(f_{L-1}(f_{L-2}(\cdots f_1(\boldsymbol{x})\cdots)))$$

How to Fit (Train) Neural Networks

- Suppose we have the training samples $(\boldsymbol{x}^{(1)}, \boldsymbol{y}^{(1)}), \cdots, (\boldsymbol{x}^{(N)}, \boldsymbol{y}^{(N)})$
- For **regression** problems we normally skip an activation in the last layer and try to solve the following minimization

$$\min_{\boldsymbol{W}_1, \cdots, \boldsymbol{W}_L, \boldsymbol{b}_1, \cdots \boldsymbol{b}_L} \frac{1}{N} \sum_{n=1}^{N} \left\| \boldsymbol{y}^{(n)} - \mathcal{M}\left(\boldsymbol{x}^{(n)}\right) \right\|^2$$

- For **classification** problems we use a soft-max in the last layer. Suppose having K classes, then $\boldsymbol{y}^{(n)}$ are vectors of length K, where for each sample the corresponding index is active

$$\min_{\boldsymbol{W}_1, \cdots, \boldsymbol{W}_L, \boldsymbol{b}_1, \cdots \boldsymbol{b}_L} \frac{1}{N} \sum_{n=1}^{N} \mathcal{H}\left(\boldsymbol{y}^{n}, \mathcal{M}\left(\boldsymbol{x}^{(n)}\right)\right)$$

- The central objective is the cross-entropy, for $\boldsymbol{y}$ and $\boldsymbol{y}'$ of length K:

$$\mathcal{H}\left(\boldsymbol{y}, \boldsymbol{y}'\right) = -\sum_{k=1}^{K} y_k \log y'_k$$

Gradient Descent for Minimization

- We saw that our fitting problem boils down to a minimization problem

$$\min_{\boldsymbol{p}} \ \mathcal{C}(\boldsymbol{p})$$

in our case $\boldsymbol{p}$ includes all the unknowns $\boldsymbol{W}_1, \cdots, \boldsymbol{W}_L, \boldsymbol{b}_1, \cdots \boldsymbol{b}_L$ and $\mathcal{C}$ is either one of the objectives in the previous slide

- Assuming $\boldsymbol{p} \in \mathbb{R}^L$, a numerical way of minimization is to start from a point $\boldsymbol{p}^{(0)}$ and iteratively perform the following steps

$$\boldsymbol{p}^{k+1} = \boldsymbol{p}^{(k)} - \eta \nabla \mathcal{C}\Big|_{\boldsymbol{p}=\boldsymbol{p}^{(k)}} \quad \text{where} \quad \nabla \mathcal{C} = \begin{pmatrix} \partial \mathcal{C}/\partial p_1 \\ \partial \mathcal{C}/\partial p_2 \\ \vdots \\ \partial \mathcal{C}/\partial p_L \end{pmatrix}$$

parameter η is called the **learning rate**

- Let's go through a simple example to see how gradient descent works (see the MATLAB code and the next slide)

Gradient Descent for Minimization

- Please refer to the MATLAB `gradientDescent.m` script
- Lets consider the very simple objective

$$\mathcal{C}(p_1, p_2) = (1 - p_1)^2 + (1 - p_2)^2 - 2\exp(-3p_1^2 - 3p_2^2)$$

  The gradient can be calculated as

$$\nabla\mathcal{C} = \begin{pmatrix} 2(p_1 - 1) + 12p_1 \exp(-3p_1^2 - 3p_2^2) \\ 2(p_2 - 1) + 12p_2 \exp(-3p_1^2 - 3p_2^2) \end{pmatrix}$$

- We can see that this objective has multiple local minimizers (two)
- Depending on where we start from we may land in either one
- A too small LR (learning rate) can make the minimization slow
- A too large LR can also make it slow or never converging!
- LR can affect which minimizer we converge to, but this is beyond our control

What is Stochastic Gradient Descent?

- Recall when we had N training samples $(\boldsymbol{x}^{(1)}, \boldsymbol{y}^{(1)}), \cdots, (\boldsymbol{x}^{(N)}, \boldsymbol{y}^{(N)})$ our fitting objective was in one of the forms:

$$\min_{\boldsymbol{p}} \; \frac{1}{N}\sum_{n=1}^{N} \left\| \boldsymbol{y}^{(n)} - \mathcal{M}_{\boldsymbol{p}}\left(\boldsymbol{x}^{(n)}\right) \right\|^2 \qquad \min_{\boldsymbol{p}} \; \frac{1}{N}\sum_{n=1}^{N} \mathcal{H}\left(\boldsymbol{y}^{n}, \mathcal{M}_{\boldsymbol{p}}\left(\boldsymbol{x}^{(n)}\right)\right)$$

  Here $\boldsymbol{p}$ is a hyper parameter representing all our unknowns $\boldsymbol{W}_1, \cdots, \boldsymbol{W}_L, \boldsymbol{b}_1, \cdots \boldsymbol{b}_L$.
- In other words, we are interested in objectives of the form

$$\mathcal{C}(\boldsymbol{p}) = \frac{1}{N}\sum_{n=1}^{N} \mathcal{C}_n(\boldsymbol{p}),$$

  where $\mathcal{C}_n$ only depends on the sample $(\boldsymbol{x}^{(n)}, \boldsymbol{y}^{(n)})$.
- Notice that to calculate $\nabla \mathcal{C}$ we need to calculate N gradients

$$\nabla \mathcal{C}(\boldsymbol{p}) = \frac{1}{N}\sum_{n=1}^{N} \nabla \mathcal{C}_n(\boldsymbol{p})$$

What is Stochastic Gradient Descent?

$$\nabla \mathcal{C}(\boldsymbol{p}) = \frac{1}{N} \sum_{n=1}^{N} \nabla \mathcal{C}_n(\boldsymbol{p})$$

- Since gradient calculation can be computationally expensive, in stochastic GD, at each minimization iteration we pick a **random** subset of all the samples $B \subset \{1, 2, \cdots, N\}$ and use it as an approximation of the gradient

$$\nabla \mathcal{C}(\boldsymbol{p}) \approx \frac{1}{\text{size }(B)} \sum_{n \in B} \nabla \mathcal{C}_n(\boldsymbol{p})$$

- If B is too small our gradient approximation may be too off!
- On the other hand large B may require a lot of gradient calculations
- Usually, after selecting the batch size, N_B we split our N data samples into N/N_B batches and in each GD iteration use one batch
- Each SGD **iteration** goes through one batch. Each **epoch** indicates going through the whole training set

Back Propagation

- This is another terminology that you probably hear a lot in deep learning
- Recall that you had to calculate the derivative with respect to each sample and each sample function is a complicated nested function, e.g.,

$$C_n = \left\| \boldsymbol{y}^{(n)} - f_L(f_{L-1}(f_{L-2}(\cdots f_1(\boldsymbol{x}_n)\cdots))) \right\|^2, \quad f_\ell(\boldsymbol{z}) = \sigma_\ell \left(\boldsymbol{W}_\ell^\top \boldsymbol{z} + \boldsymbol{b}_\ell \right)$$

- Back propagation is simply the application of the chain rule to calculate the derivative of nested functions like C_n in terms of all the unknown parameters $\boldsymbol{W}_1, \cdots, \boldsymbol{W}_L, \boldsymbol{b}_1, \cdots \boldsymbol{b}_L$
- Since the actual story goes through a lot of indexing complications, let me explain things via a simple example

Back Propagation

- You can skip this example if don't have time
- Find the derivative of the following function at $w = 2$:

$$f(w) = \big(\sin\big(w^2 + 1\big)\big)^2$$

- **Method 1:** go through the pain of calculating the derivative and find

$$f'(w) = 4w\cos(w^2 + 1)\sin(w^2 + 1)$$

- Then plug in $w = 2$ to get

$$f'(2) = 8\cos(5)\sin(5)$$

Back Propagation

- Find the derivative of the following function at $w = 2$:

$$f(w) = \left(\sin\left(w^2 + 1\right)\right)^2$$

- **Method 2:** Notice that

$f = g_1(g_2(g_3(w)))$, where $g_1(g_2) = g_2^2,\ g_2(g_3) = sin(g_3),\ g_3(w) = w^2 + 1$

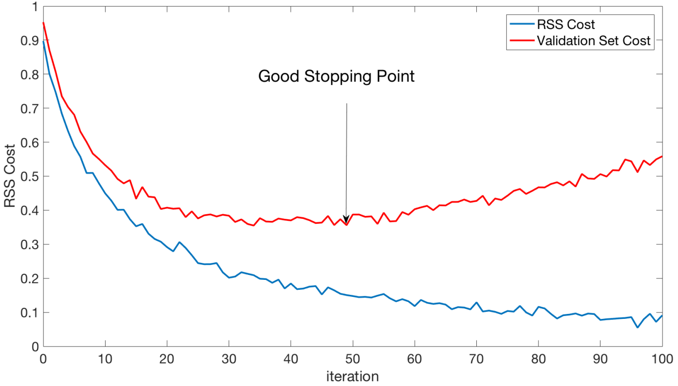
and use the chain rule

$$\frac{\partial f}{\partial w} = \frac{\partial g_1}{\partial w} = \frac{\partial g_1}{\partial g_2}\frac{\partial g_2}{\partial g_3}\frac{\partial g_3}{\partial w} = 2\sin(5) \times \cos(5) \times 4$$

- Still want to learn more about back propagation? Try these videos:
  - `https://www.youtube.com/watch?v=Ilg3gGewQ5U`
  - `https://www.youtube.com/watch?v=tIeHLnjs5U8`

Using a Validation Set to Control the Minimization

- As you observed in the previous slides gradient descent gradually decreases the RSS (or cross entropy cost) to find a minimizer
- One way to avoid overfitting, is to use a "**validation set**", independent of the training set and stop the gradient descent iterations when the validation error starts to increase

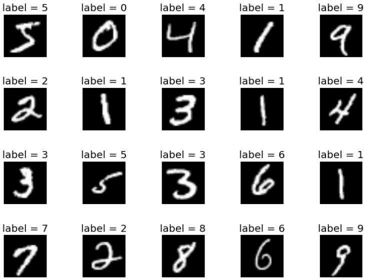


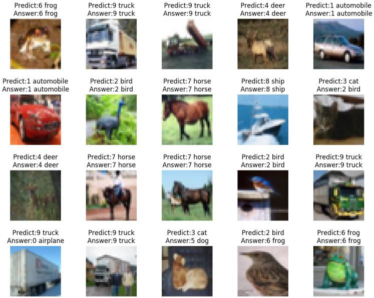
Regularization of Neural Networks to Avoid Overfitting

- Similar to linear models there are variety of techniques to avoid overfitting in neural networks
    - L_2 regularizers (similar to Ridge)
    - L_1 regularizers (Similar to LASSO)
    - Dropout and DropConnect (which drops some of the updates in the gradient descent)
        - See video: `https://www.youtube.com/watch?v=ARq74QuavAo`
        - See papers: Paper 1, Paper 2
    - Net-Trim and compression algorithms
        - See video: `https://www.youtube.com/watch?v=WxU8dp7iYg0&t`
        - See papers and code: Here
- See one implementation of NNs **in R using the H2O package**

Convolutional Neural Networks

- Deep learning has shown a lot of promise in classifying images

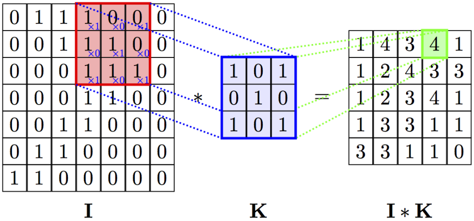




- One immediate candidate for the features is the image pixels. But if the images are too large, this would make our problem unnecessarily large
- Convolutional neural nets are a solution to this problem

Linear Filtering and Images

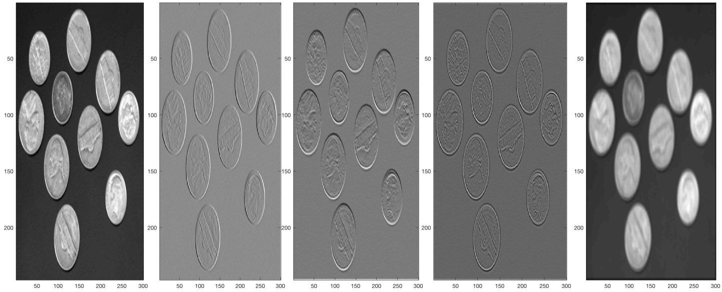
- Convolution is a linear operator widely used in image and signal processing



$$\boldsymbol{I} \star \boldsymbol{K}(m,n) = \sum_{i=1}^{M} \sum_{j=1}^{N} \boldsymbol{I}(m-i, n-j)\boldsymbol{K}(i,j)$$

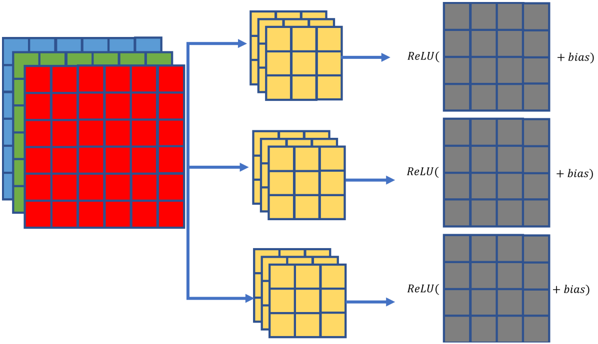
- Depending on the type of filter we pick for $\boldsymbol{K}$ the output image can have different properties (blurred, sharpened, edges detected, etc)
- See the MATLAB code `convExample.m`

Example of Image Convolution with Different Kernels



- If the filters are selected wisely, their output can be considered as alternative features to pixels
- In a CNN, we let the neural network learn these filters! In other words, CNN wisely chooses the right features that are the best for prediction
- For color images (RGB) we can have 3D filters each filter applicable to one channel

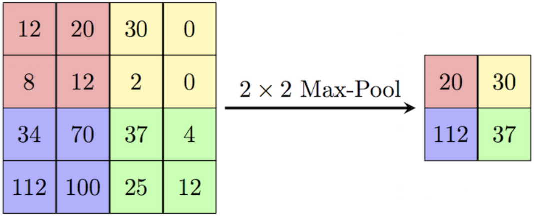
Convolutional Layers



- We can define as many 2D or 3D convolutional filters (here 3 3D filters of size $3 \times 3 \times 3$)
- The total number of parameters that need to be learnt for this layer is going to be $3 \times (27 + 1)$
- An input image of $6 \times 6 \times 3$ is mapped to a tensor of size $4 \times 4 \times 3$

Max Pooling

- Is another operation that allows us to reduce the input size by taking a max operation over smaller windows across the image



- To help you better understand the topic, we will consider an implementation of the CNN that learns to read handwritten digits (MNIST data set) — see the Python code

Recurrent Neural Networks

- While CNNs work quite promising for images, they may not be the best modeling tools for other data sets such as time series data
- For time series data and stream inputs (e.g., text analytics), **recurrent neural networks (RNNs)** are of major attention
- Remember in standard neural network the output of the hidden layer was in the form

$$\boldsymbol{h} = \sigma\left(\boldsymbol{W}_x^\top \boldsymbol{x} + \boldsymbol{b}\right)$$

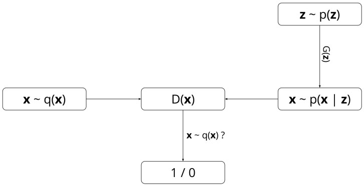
- In RNNs the input is stream $\boldsymbol{x}_t$ and we have another coefficient matrix that makes the current hidden output dependent on the previous one:

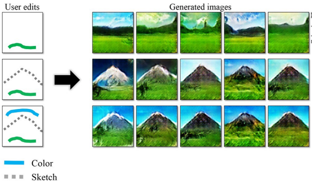
$$\boldsymbol{h}_t = \sigma\left(\boldsymbol{W}_x^\top \boldsymbol{x}_t + \boldsymbol{W}_h^\top \boldsymbol{h}_{t-1} + \boldsymbol{b}\right)$$

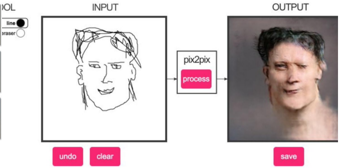
- To learn more and see some cool applications see: `https://www.youtube.com/watch?v=6niqTuYFZLQ&t=1850s`

Generative Adversarial Networks

- Is the most recent breakthrough in machine learning started in 2015
- Basically once we pass enough samples to a GAN network, it starts to learn how to generate similar samples







- To learn more and see some interesting applications see: This Video ; or This Video

Questions?

References

http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture10.pdf, 2016.

Introduction to GANs: https://arxiv.org/pdf/1701.00160.pdf, 2016.

J. Friedman, T. Hastie, and R. Tibshirani.
The elements of statistical learning.
Springer series in statistics, 2nd edition, 2009.

I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio.
Deep learning, volume 1.
MIT press Cambridge, 2016,
link:http://www.deeplearningbook.org/contents/convnets.html.