IFI 9000 Analytics Methods
Neural Networks & Deep Learning

by **Houping Xiao**

Spring 2021

Introduction

We are going to cover ...

- Matrix notation for linear models, especially multi-ouput models
- Structure of the brain
- Neural network models in matrix form
- Gradient descent technique for minimization
- NN fitting objective and (stochastic) gradient descent
- Introduction to signal processing and linear filtering
- Convolutional neural network architectures
- Other variants of NNs, Recurrent NNs, U-Nets, etc

Vectors and Matrices

- A vector is a one dimensional array of numbers

$$\boldsymbol{a} = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} \in \mathbb{R}^n, \quad \text{e.g.} \quad \boldsymbol{a} = \begin{pmatrix} 0 \\ 1 \\ 2 \\ 3 \end{pmatrix} \in \mathbb{R}^4.$$

- A matrix is a 2-dimension array of numbers

$$\boldsymbol{A}_{m \times n} = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ a_{21} & \cdots & a_{2n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix} \in \mathbb{R}^{m \times n}, \quad \text{e.g.} \quad \boldsymbol{A} = \begin{pmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{pmatrix} \in \mathbb{R}^{3 \times 3}.$$

Matrix transpose and product

- Transpose of a matrix:

$$
\boldsymbol{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}, \quad \boldsymbol{A}^\top = \begin{pmatrix} a_{11} & a_{21} & \cdots & a_{m1} \\ a_{12} & a_{22} & \cdots & a_{m2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \cdots & a_{mn} \end{pmatrix}.
$$

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

$$
\begin{pmatrix} a_{11} & \cdots & a_{1n} \\ a_{21} & \cdots & a_{2n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix} \begin{pmatrix} b_{11} & \cdots & b_{1n} \\ b_{21} & \cdots & b_{2n} \\ \vdots & \ddots & \vdots \\ b_{m1} & \cdots & b_{mn} \end{pmatrix} = \begin{pmatrix} \sum_{k=1}^m a_{1k} b_{k1} & \cdots & \sum_{k=1}^m a_{1k} b_{kp} \\ \sum_{k=1}^m a_{2k} b_{k1} & \cdots & \sum_{k=1}^m a_{2k} b_{kp} \\ \vdots & \ddots & \vdots \\ \sum_{k=1}^m a_{nk} b_{k1} & \cdots & \sum_{k=1}^m a_{nk} b_{kp} \end{pmatrix}
$$

Linear models in matrix form

- Typically, vectors and matrices are denoted as lowercase uppercase bold letters, respectively
- We have seen that

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

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

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

Multi-response linear models in matrix form

- 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

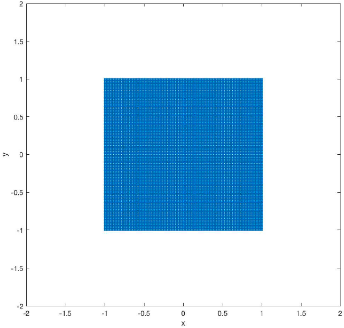
$$\boldsymbol{y} = \boldsymbol{b} + \boldsymbol{W}\boldsymbol{x}$$

where

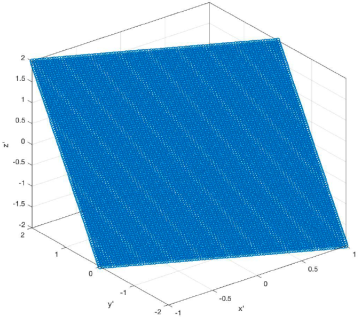
$\boldsymbol{b}$ $\boldsymbol{W}$ $\boldsymbol{x}$

Matrices as a way of linear transformation

$$\begin{pmatrix} x'_1 & x'_2 & \cdots & x'_n \\ y'_1 & y'_2 & \cdots & y'_n \\ z'_1 & z'_2 & \cdots & z'_n \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ -1 & 1 \\ 0 & 2 \end{pmatrix} \begin{pmatrix} x_1 & x_2 & \cdots & x_n \\ y_1 & y_2 & \cdots & y_n \end{pmatrix}$$

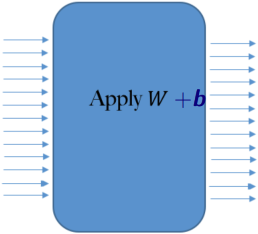


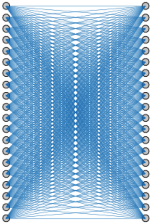
$\Rightarrow$



Fitting multi-response linear models

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

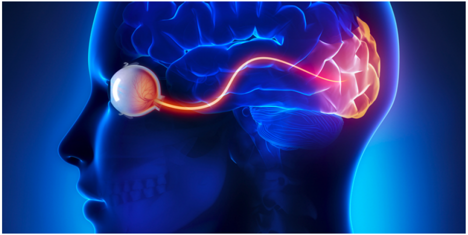


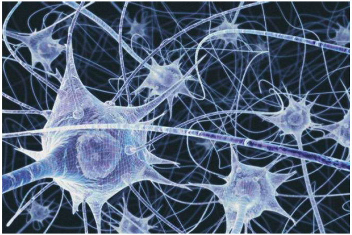


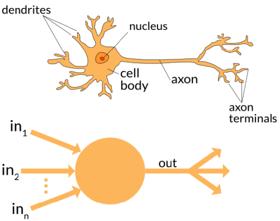
- Suppose we have the training samples $(\boldsymbol{x}_1, \boldsymbol{y}_1), \cdots, (\boldsymbol{x}_N, \boldsymbol{y}_N)$. To fit the model to the training data, we only need to miniize the following RSS:

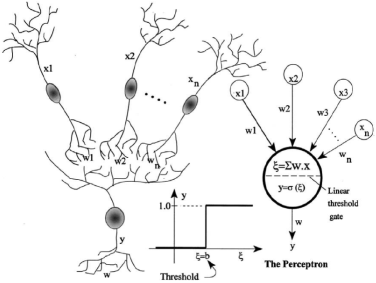
$$\underset{\boldsymbol{W}, \boldsymbol{b}}{\textit{minimize}} \quad \sum_{i=1}^{N} ||\boldsymbol{y}_i - \boldsymbol{b} - \boldsymbol{W}\boldsymbol{x}_i||^2$$

Structure of the brain









Nonlinear activation applied to a vector

- Sigmoid function

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

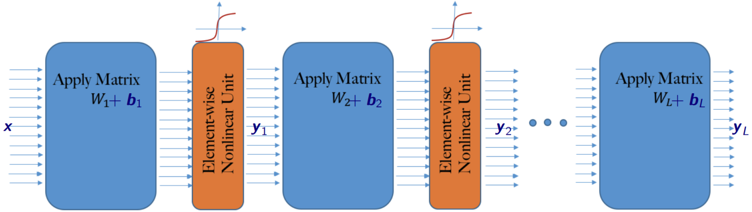
- when sigmoid function is applied to a vector or a matrix, it applies to each component individually

$$\sigma\left(\begin{bmatrix} 0 \\ 1 \\ 2 \end{bmatrix}\right) = \begin{bmatrix} \frac{e^0}{1+e^0} \\ \frac{e^1}{1+e^1} \\ \frac{e^2}{1+e^2} \end{bmatrix}$$

- Another widely used activation is the rectified linear unit (ReLU):

$$ReLU(x) = \max(x, 0)$$

Standard architecture of neural networks



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

$$\boldsymbol{y}_1 = \sigma_1(\boldsymbol{W}_1 \boldsymbol{x} + \boldsymbol{b}_1)$$

$$\boldsymbol{y}_2 = \sigma_2(\boldsymbol{W}_2 \boldsymbol{y}_1 + \boldsymbol{b}_2)$$

$$\vdots$$

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

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.,

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

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

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

- Example:

$$\begin{pmatrix} 0.5 \\ 1.8 \\ -2.3 \\ 0.9 \\ 0.3 \end{pmatrix} \underset{\rightarrow}{\text{soft-max}} \begin{pmatrix} 0.14 \\ 0.52 \\ 0.01 \\ 0.21 \\ 0.12 \end{pmatrix}$$

How to train a neural network

- 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

$$f_l(\boldsymbol{z}) = \sigma_l(\boldsymbol{W}_l \underline{z} + \boldsymbol{b}_l),$$

  then we have

$$\begin{aligned} \boldsymbol{y}_L &= f_L(\boldsymbol{y}_{L-1}) \\ &= f_L(f_{L-1}(\boldsymbol{y}_{L-2})) \\ &\vdots \\ &= f_L(f_{L-1}(f_{L-2}(_1(\boldsymbol{x})\cdots))) \end{aligned}$$

- Basically,

$$\boldsymbol{y}_L = \mathcal{M}(\boldsymbol{x}), \quad \text{where} \quad \mathcal{M}(\boldsymbol{x}) = f_L(f_{L-1}(f_{L-2}(_1(\boldsymbol{x})\cdots)))$$

How to train a neural network

- 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

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

- 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

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

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

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

Gradient descent for minimization

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

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

  in our case $\boldsymbol{p}$ includes all the unknown $bmW_1, \cdots, \boldsymbol{W}_L, \boldsymbol{b}_1, \cdots, \boldsymbol{b}_L$ 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

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

  η is called the **learning rate**
- Let's go through a simple example to review how gradient descent works

Gradient descent for minimization

- 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 calcualted 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 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 beyound our control

OVO is usually more reliable, however for large number of classes OVA is computationally more desirable

Review 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:

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

  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 $\triangledown \mathcal{C}$ we need to calculate N gradients

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

Review stochastic gradient descent

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

- 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

$$\bigtriangledown\mathcal{C}(\boldsymbol{p}) \approx \frac{1}{|B|}\sum_{n\in B}\bigtriangledown\mathcal{C}_n(\boldsymbol{p})$$

- 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.,

$$C_n = \left|\left|\boldsymbol{y}^{(n)} - f_L(f_{L-1}(f_{L-2}(_1(\boldsymbol{x})\cdots)))\right|\right|^2, \quad , f_l(\boldsymbol{z}) = \sigma_l(\boldsymbol{W}_l\boldsymbol{z} + \boldsymbol{b}_l)$$

- 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, chain rule simple example

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

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

- Solution: Notice that

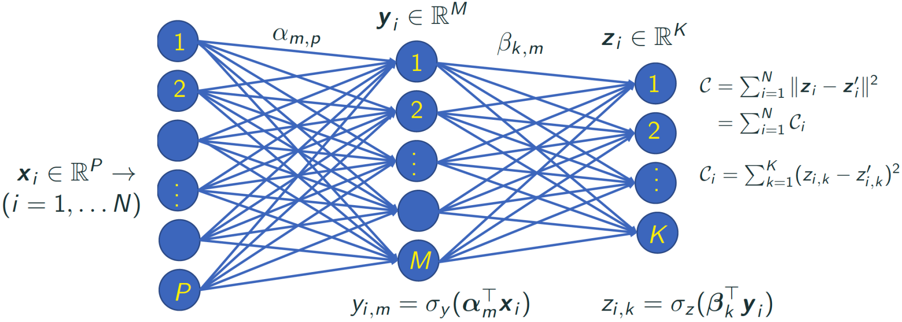
$$f = g_1(g_2(g_3(w))); \quad g_1(g_2) = g_2, g_2(g_3) = sin(g_3), g_3(w) = w^2 + 1$$

and use the chain rule

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

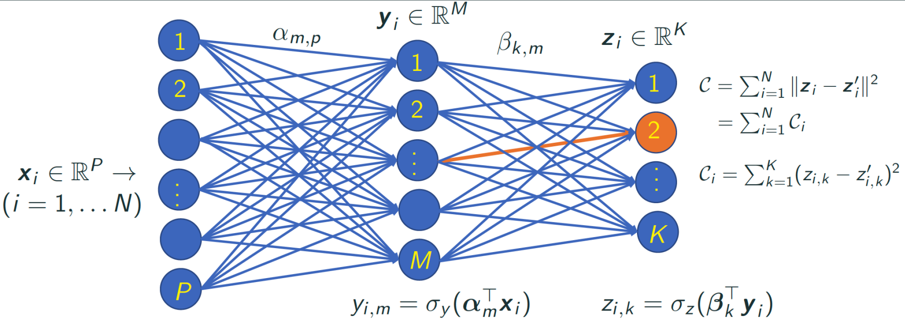
- Some useful videos about back propagation:
  - https://www.youtube.com/watch?v=Ilg3gGewQ5U
  - https://www.youtube.com/watch?v=tIeHLnjs5U8

Back propagation



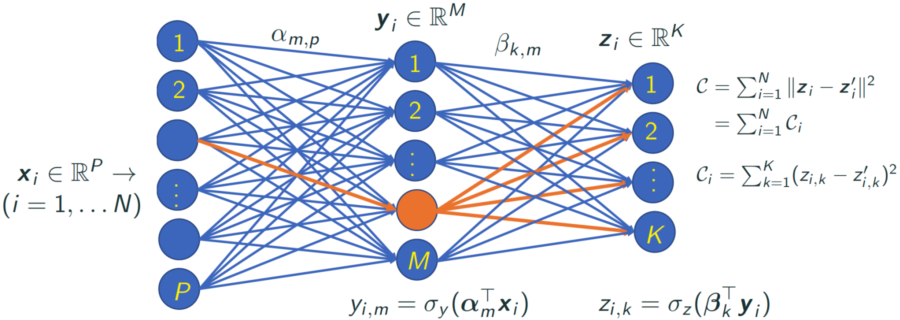
- Use chain rule to derive

$$\frac{\partial \mathcal{C}_i}{\partial \beta_{k_0,m_0}}, \frac{\partial \mathcal{C}_i}{\partial \alpha_{m_0,p_0}}$$



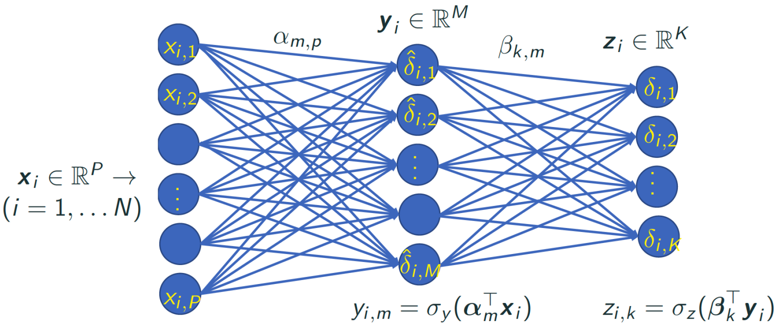
- Last layer sensitivity:

$$
\begin{aligned}
\frac{\partial \mathcal{C}_i}{\partial \beta_{k_0,m_0}} &= \frac{\partial \mathcal{C}_i}{\partial z_{i,k_0}} \frac{\partial z_{i,k_0}}{\partial \beta_{k_0,m_0}} \\
&= 2(\sigma_z(\beta_{k_0}^\top \boldsymbol{y}_i) - z'_{i,k_0}) \sigma'_z(\beta_{k_0}^\top \boldsymbol{y}_i) y_{i,m_0} \\
&= \delta_{i,k_0} y_{i,m_0}
\end{aligned}
$$



- Other layers sensitivity:

$$
\begin{aligned}
\frac{\partial \mathcal{C}_i}{\partial \alpha k_0, m_0} &= \sum_{k=1}^K \frac{\partial \mathcal{C}_i}{\partial z_{i,k}} \frac{\partial z_{i,k}}{\partial y_{i,m_0}} \frac{\partial y_{i,m_0}}{\partial \alpha_{k_0,m_0}} \\
&= \sum_{k=1}^K 2(\sigma_z(\beta_k^\top \mathbf{y}_i) - z_{i,k}')\sigma_z'(\beta_k^\top \mathbf{y}_i)\beta_{k,m_0}\sigma_y'(\alpha_{m_0}^\top \mathbf{x}_i)x_{i,p_0} \\
&= \sigma_y'(\alpha_{m_0}^\top \mathbf{x}_i)\left(\sum_{k=1}^K \delta_{i,k}\beta_{k,m_0}\right) x_{i,p_0} = \hat{\delta}_{i,m_0} x_{i,p_0}
\end{aligned}
$$

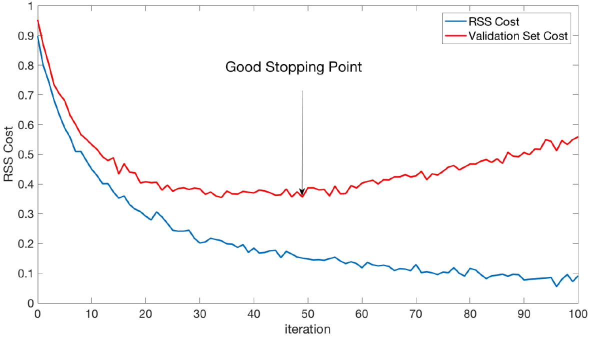


- Sensitivity summary:

$$\frac{\partial \mathcal{C}_i}{\partial \beta_{k_0,m_0}} = \delta_{i,k_0} y_{i,m_0}, \qquad \frac{\partial \mathcal{C}_i}{\partial \alpha_{m_0,p_0}} = \hat{\delta}_{i,m_0} x_{i,p_0}$$

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 over-fitting, is to use a "**validation set**", independent of the training set and stop the gradient descent iterations when the validation error starts to increase

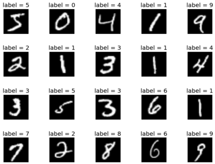


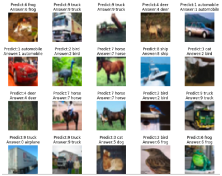
Regularization of neural networks to avoid overfitting

- Similar to linear models there are variety of techniques to avoid over-fitting in neural networks
  - L2 regularizers (similar to Ridge)
  - L1 regularizers (Similar to LASSO)
  - Dropout
    - See video: https://www.youtube.com/watch?v=ARq74QuavAo
    - See papers: Paper 1

Convolutional neural networks

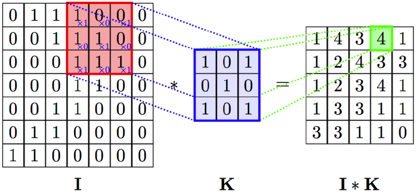
- Deep learning has shown a lot of promise in classifying images





Linear filtering and images

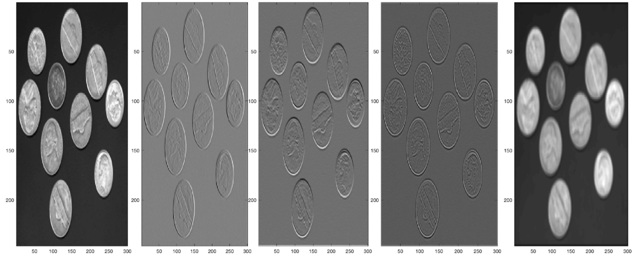
- Convolution is a linear operator widely used in image and signal processing



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

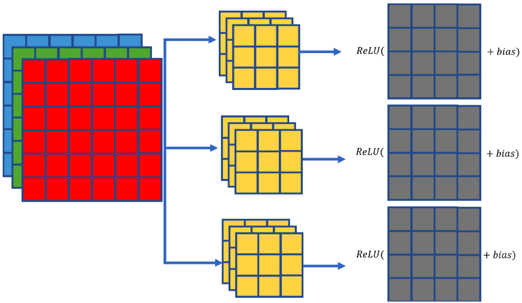
- Depending on the type of filter we pick for K the output image can have different properties (blurred, sharpened, edges detected, etc)

Examples of image convolution with different kernels



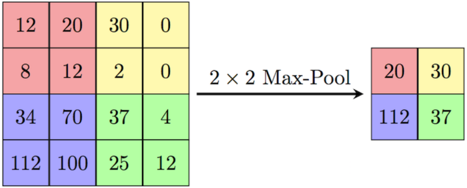
- If the filters are selected wisely, their output can be considered as alternative features to pixels
- In a CNN, we let the neural network learn these filters! In other words, CNN wisely chooses the right features that are the best for prediction
- For color images (RGB) we can have 3D filters each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

Convolutional layers



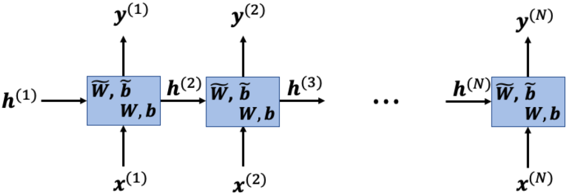
- We can define as many 2D or 3D convolutional filters (here 3 3D filters of size $3 \times 3 \times 3$)
- The total number of parameters that need to be learnt for this layer is going to be $3 \times (27 + 1)$
- An input image of $6 \times 6 \times 3$ is mapped to a tensor of size $4 \times 4 \times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

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



Recurrent neural networks

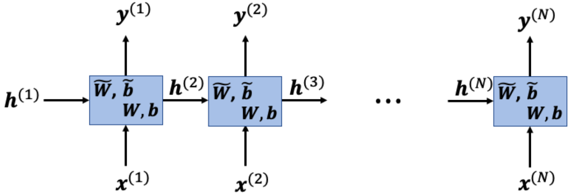
- While CNNs work quite promising for images, they may not be the best modeling tools for other data sets such as time series data
- For temporal, or time-series data and stream inputs (e.g., text streams), recurrent neural networks (RNNs) are of major attention



- We assume a sequence of data is streamed as N time instances, and mapped to a sequence of response (here of the same length).
- For now let's assume that the input and output have similar lengths

RNN: governing equations

- Remember in standard neural network the output of the hidden layer was in the form $\boldsymbol{h} = \sigma(\boldsymbol{W}\boldsymbol{x}x + \boldsymbol{b})$



- In RNNs the input is a stream x(t) and we have another coefficient matrix that makes the current hidden output dependent on the previous one:

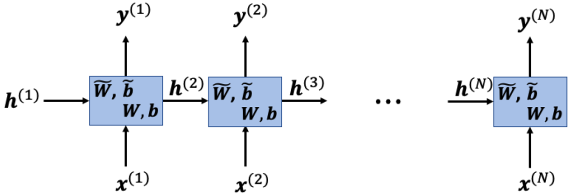
$$\boldsymbol{h}^{(t)} = \sigma\left(\boldsymbol{W}\begin{pmatrix} \boldsymbol{h}^{(t-1)} \\ \boldsymbol{x}^{(t-1)} \end{pmatrix} + \boldsymbol{b}\right),$$

$$\boldsymbol{y}^{(t)} = \sigma\left(\tilde{\boldsymbol{W}}\boldsymbol{h}^{(t)} + \tilde{\boldsymbol{b}}\right), t = 1, \cdots, N$$

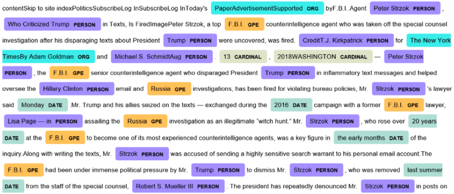
- Training cost per sample: $\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = \sum_{t=1}^{N} L\left(\boldsymbol{y}^{(t)}, \hat{\boldsymbol{y}}^{(t)}\right)$

Types of RNN and applications

- The following architecture is many-to-many, with the input and output having the same length

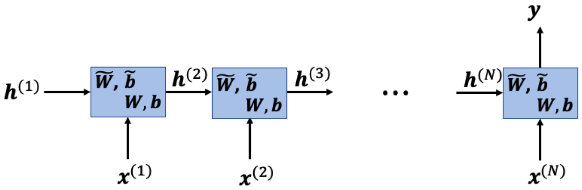


- Application example is named-entity recognition (classify unstructured text into predefined classes)



Types of RNN and applications

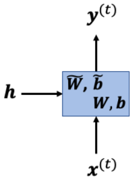
- The following architecture is many-to-one



- Application example is sentiment classification (review systems, scoring systems)

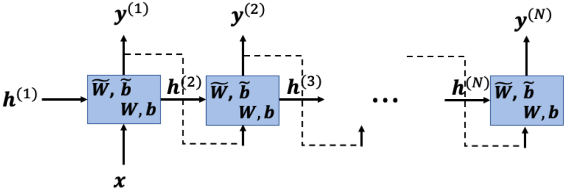
Types of RNN and applications

- The following architecture is one-to-one



- This is somehow equivalent to traditional one-layer network (real-time mapping)

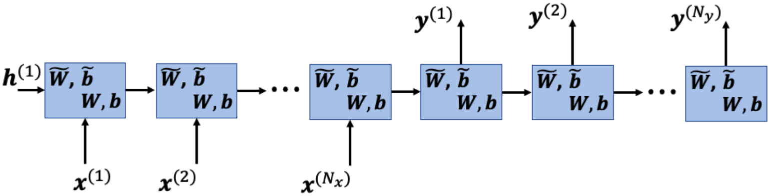
- The following architecture is one-to-many



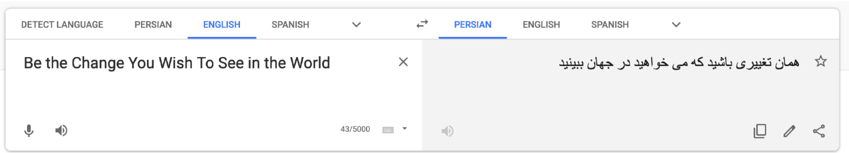
- Application example is music generation

Types of RNN and applications

- The following architecture is many-to-many, with the input and output having different lengths

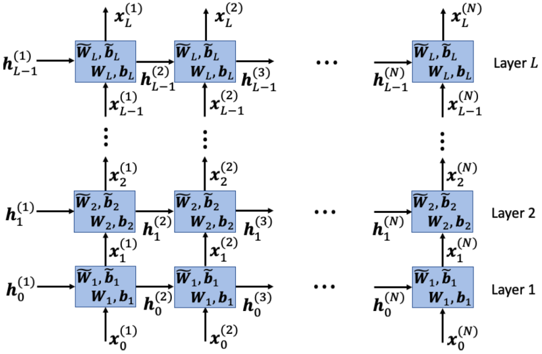


- Application example is machine translation



Deep RNNs

- All the architectures we explained so far can become deep and layered



- In practice we do not need very deep RNNs (unlike standard DNNs which can be very deep)

- One hot encoding is normally used to convert a vocabulary into digital inputs

1. Me
2. Water
3. Food
4. Cave
5. Go
6. Dinosaur
7. Sleep
8. Stone
9. Hunting
10. Stick

Me	**Go**	**Hunting**
1	0	0
0	0	0
0	0	0
0	0	0
0	1	0
0	0	0
0	0	0
0	0	0
0	0	1
0	0	0

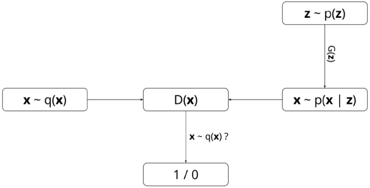
- It is normally easier and more robust to do the one hot encoding with the words other than letters

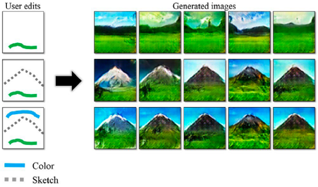
Problems with standard RNNs and remedies

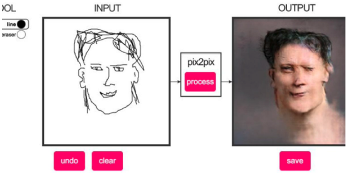
- Hard to train and vanishing gradient
- Difficulty accessing information from long time ago
- Two main variants of RNNs:
  - Long Short-Term Memory (LSTM)
  - Gated Recurrent Units (GRUs)
- To learn more and see some cool applications see: https://www.youtube.com/watch?v=6niqTuYFZLQt=1850s

Deep RNNs

- 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

The End