Day 6: Neural Networks

Summer STEM: Machine Learning

Department of Electrical and Computer Engineering
NYU Tandon School of Engineering
Brooklyn, New York

Outline

Machine Learning Problem Pipeline

1. Formulate the problem: regression, classification, or others?
2. Gather and visualize the data
3. Design the model and the loss function
4. Train your model
    (a) Perform feature engineering
    (b) Construct the design matrix
    (c) Choose regularization techniques
    (d) Tune hyper-parameters using a validation set
    (e) If the performance is not satisfactory, go back to step (a).
5. Evaluate the model on a test set

Outline

Motivation

- Cannot rely on closed form solutions
  - Computation efficiency: operations like inverting a matrix is not efficient
  - For more complex problems such as neural networks, a closed-form solution is not always available
- Need an optimization technique to find an optimal solution
  - Machine learning practitioners use **gradient**-based methods

Gradient Descent Algorithm

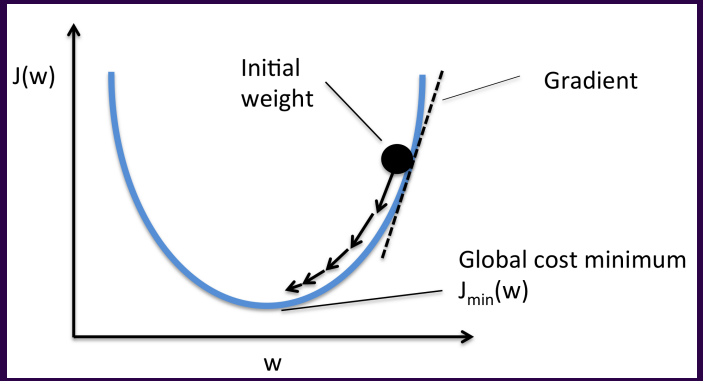
- Update Rule

  Repeat{

  $$\mathbf{w}_{new} = \mathbf{w} - \alpha \nabla J(\mathbf{w})$$

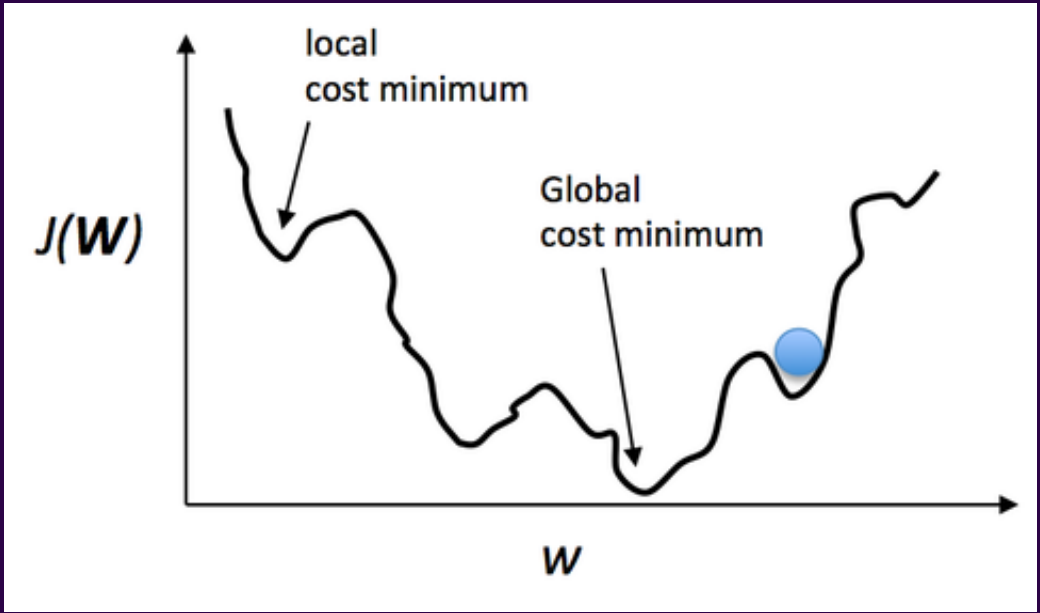
  }

  α is the learning rate

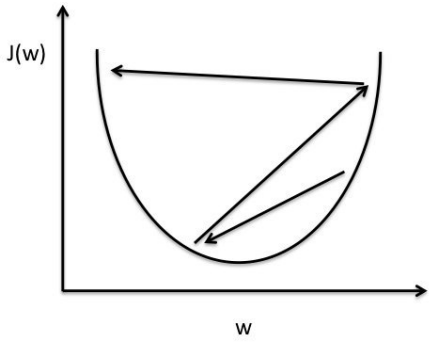


General Loss Function Contours

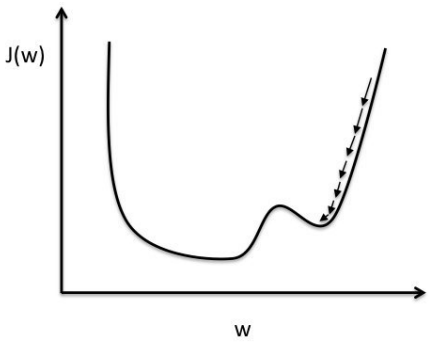
- Most loss function contours are not perfectly parabolic
- Our goal is to find a solution that is very close to global minimum by the right choice of hyper-parameters



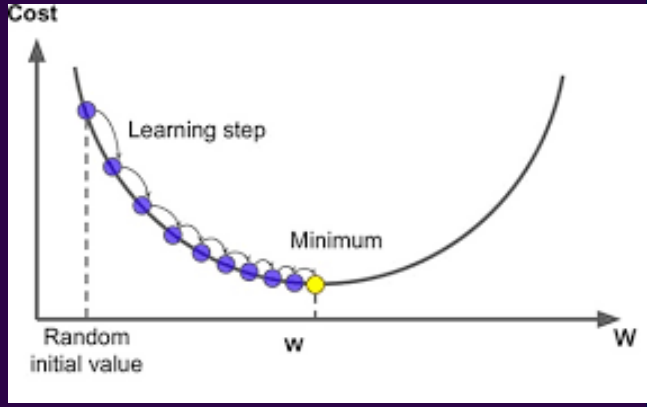
Understanding Learning Rate



Large learning rate: Overshooting.



Small learning rate: Many iterations until convergence and trapping in local minima.



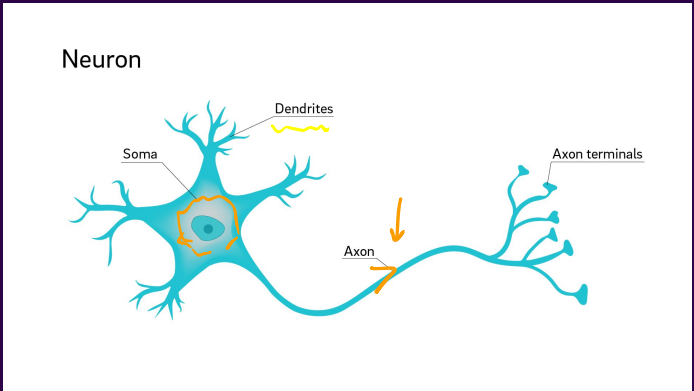
Correct learning rate

Some Animations

- Demonstrate gradient descent animation

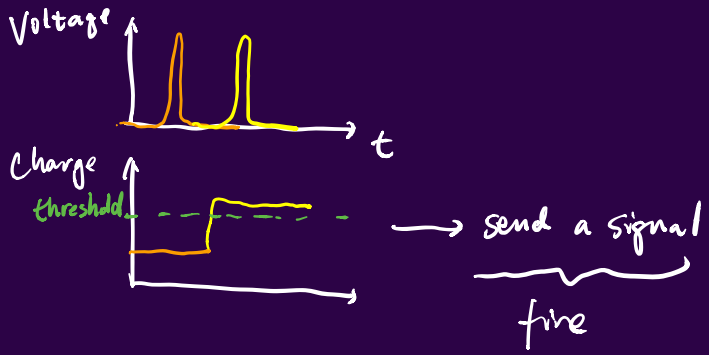
Outline

Biological Neuron



Source: David Baillot/UC San Diego

- Communicate with other cells
- A neuron can send electrochemical signals to other neurons

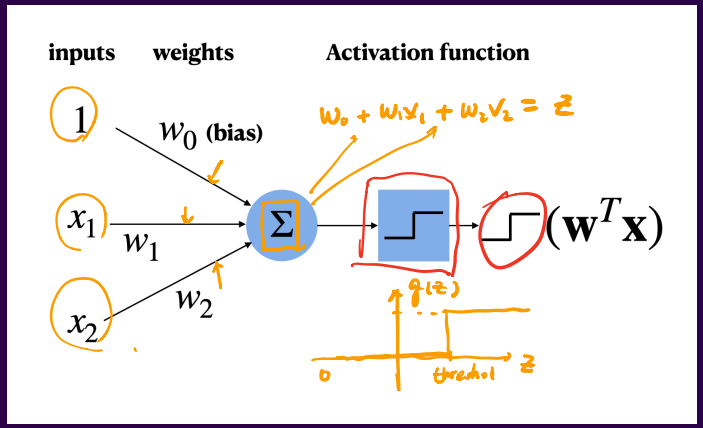


Mathematical Neuron: Perceptron

Biological Neuron:

- A neuron can receive electrochemical signals from other neurons;
- A neuron fires once its accumulated electric charge passes a certain threshold.
- Neurons that fire together wire together.

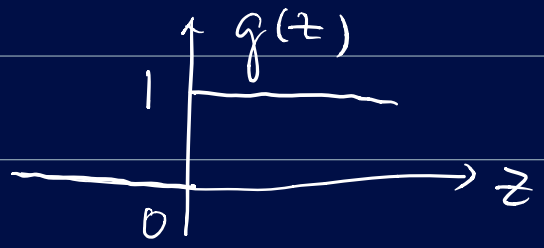
Mathematical Neuron (Perceptron)



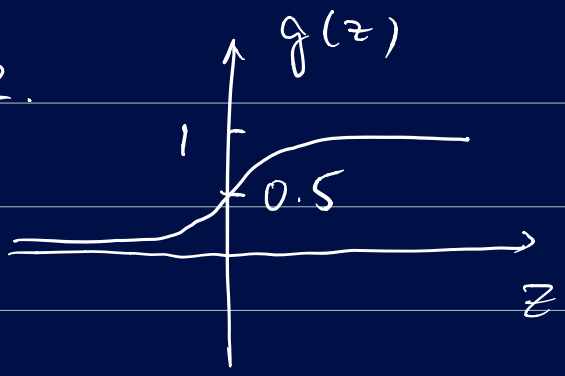
$g(w^T x)$ $\quad w = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \end{bmatrix}$

Activation function

1. Step function

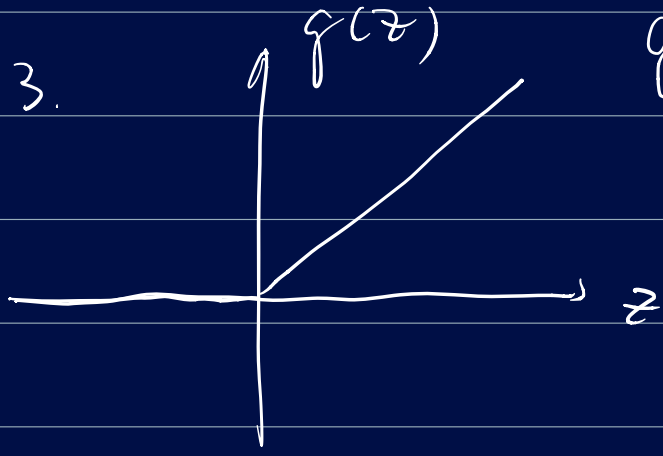


2. Sigmoid



3. $g(z) = \max(z, 0)$

ReLu



If use sigmoid activation

our model is simply

$f(x) = \sigma(w^T x)$

Relation to Logistic Regression

What if we use the sigmoid function as the activation?

$$f(\mathbf{x}) = \sigma(\mathbf{w}^T\mathbf{x})$$

Decision boundary is a line: how is this supposed to revolutionize machine learning?

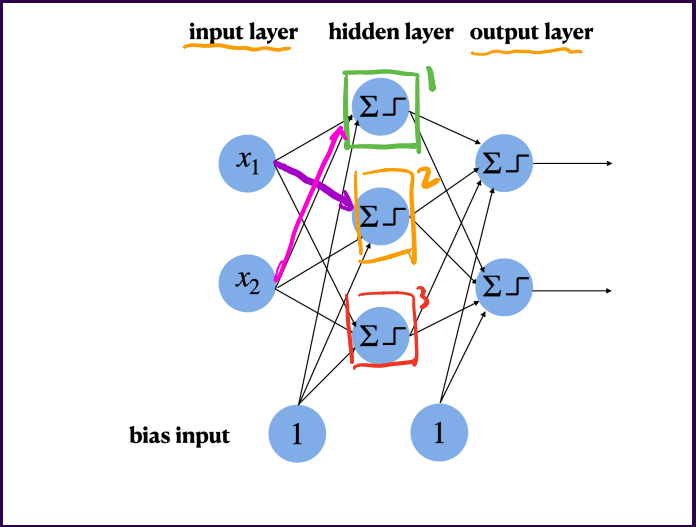
Multi-layer Perceptron (MLP)

We need more neurons and we need to connect them together!

- Many ways to do that...
- Today: multi-layer perceptron/fully connected feed-forward network.

$\sigma(w^T x)$

MLP Example



- What is the shape of the input and output?
- How many parameters does this model have?
- What activation function would you use for the output layer? Why?

w_{ij} : the weight associated with the arrow that points from x_j to the neuron i.

$$z_1 = w_{10} x_0 + w_{11} x_1 + w_{12} x_2$$

$x_0 = 1$ $$z_2 = w_{20} x_0 + w_{21} x_1 + w_{22} x_2$$

$$z_3 = w_{30} x_0 + w_{31} x_1 + w_{32} x_2$$

$$z = \begin{bmatrix} z_1 \\ z_2 \\ z_3 \end{bmatrix} = \underbrace{\begin{bmatrix} w_{10} & w_{11} & w_{12} \\ w_{20} & w_{21} & w_{22} \\ w_{30} & w_{31} & w_{32} \end{bmatrix}}_{W} \begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix}$$

Let's say we don't use any activation function

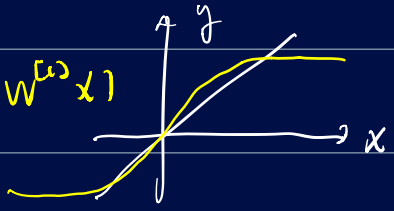
$$z = W^{[1]} x$$

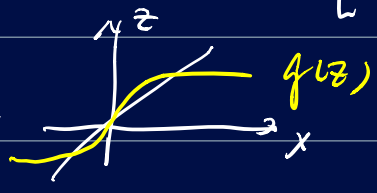
$$y = w^{[2]} g(z) = w^{[2]} \underbrace{g(W^{[1]} x)}$$

$$g(z) = \begin{bmatrix} g(z_1) \\ g(z_2) \\ \vdots \end{bmatrix}$$

$z = w^{[1]} x$

$y = w^{[2]} g(w^{[1]} x)$





More about MLPs

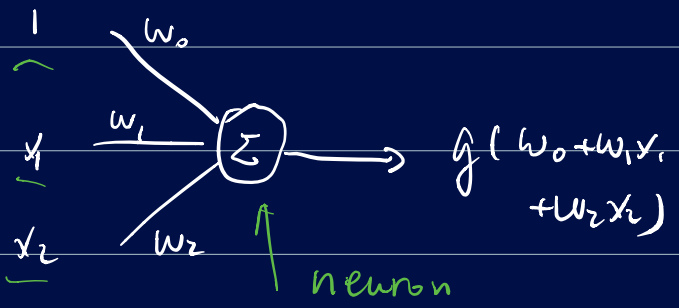
- Many choices for the activation function: Sigmoid, Tanh, ReLU, Swish, etc.
- Many choices for the number of hidden layers and the number of neurons per layer.
- MLPs can approximate any continuous function given enough data.
- MLPs can overfit, but we know many effective ways of regularization.

Exercise: TensorFlow Keras Basics

Open `demo_tf_keras_basics.ipynb`

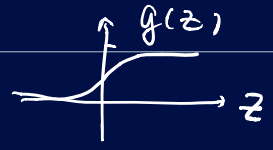
Review $x = [1\ x_1\ x_2\ \dots]^T$

- Perceptron $\sigma(w^T x)$ σ: sigmoid

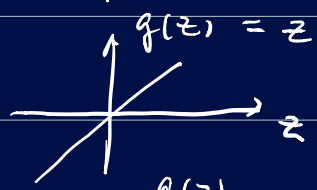


Activation functions

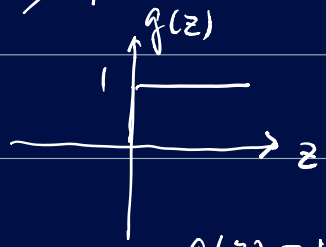
Sigmoid



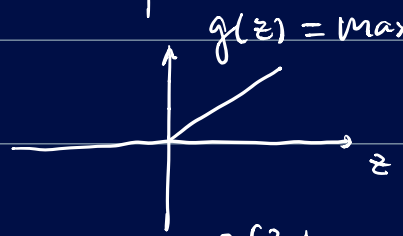
linear $g(z) = z$ identity function



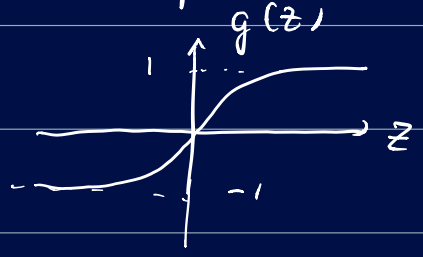
Step function



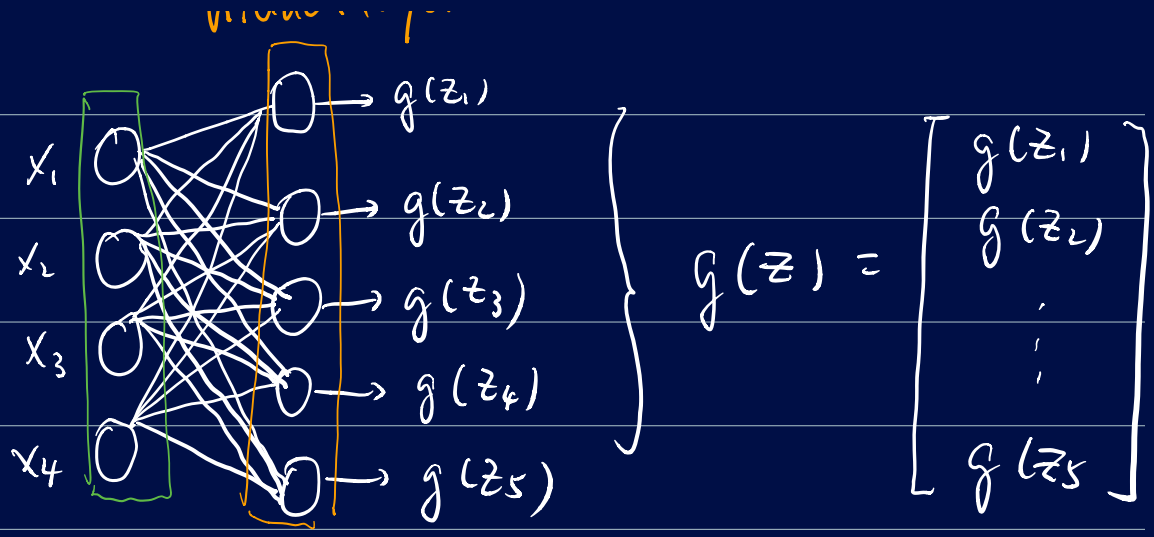
ReLu $g(z) = \max(z, 0)$



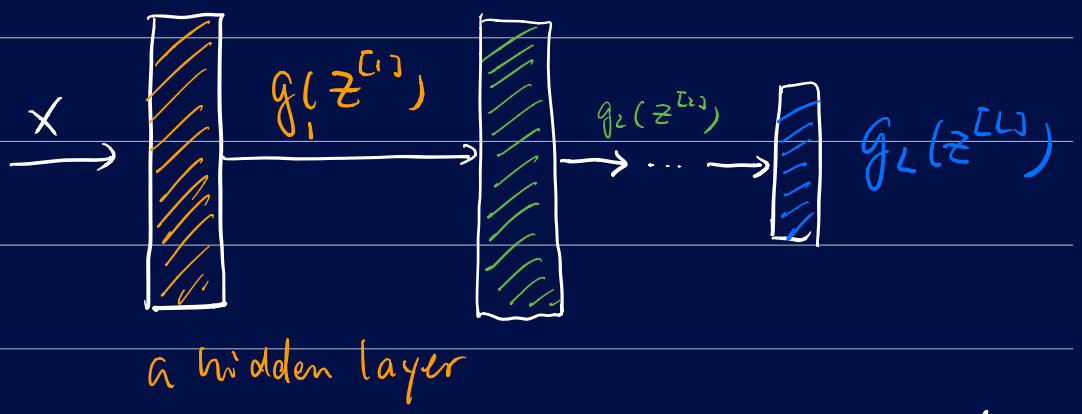
Tanh



hidden layer



$z_i = \sum w_{ij} x_j$ $\quad w_{ij}$ points from x_j to z_i



$$\hat{y} = f(x) = g_L(z^{[L]})$$

the choice of g_L depends on the task:

- regression / classification
- our knowledge about the output

Ex. • binary classification ($y = \{0, 1\}$)

g_L : Sigmoid

• multi-class classification

g_L : Softmax

• regression ($y \in \mathbb{R}$ $(-\infty, +\infty)$)

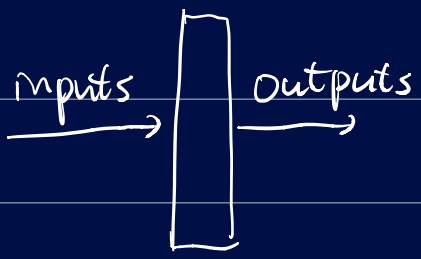
g_L : Linear

```
from tensorflow.keras.layers import Dense
from tensorflow.keras import layers
```

• fully-connected layer : layers.Dense( )

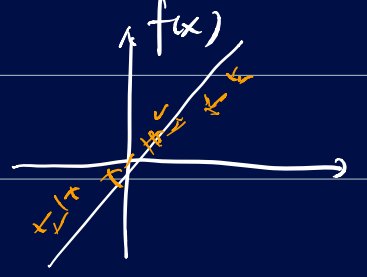
```
l1 = layers.Dense(arg1, activation="relu", input_shape=(4,))
```

arg1 : size of outputs
# of neurons



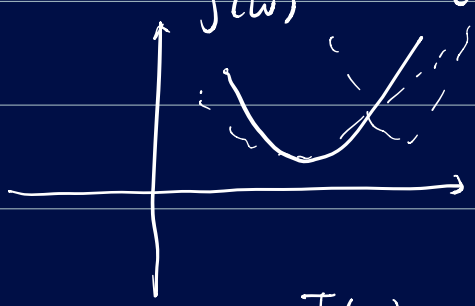
Review : Gradient Descent

Toy example : Consider scalar-valued features x, labels y, and suppose our model is $\hat{y} = f(x) = wx$ w : a scalar



$$J(w) = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

$$= \frac{1}{N}\sum_{i=1}^{N}(y_i - wx_i)^2$$



$i=1$ $\quad y_1^2 - 2wx_1y_1 + w^2x_1^2$

2 $\quad y_2^2 - 2wx_1y_1 + w^2x_2^2$

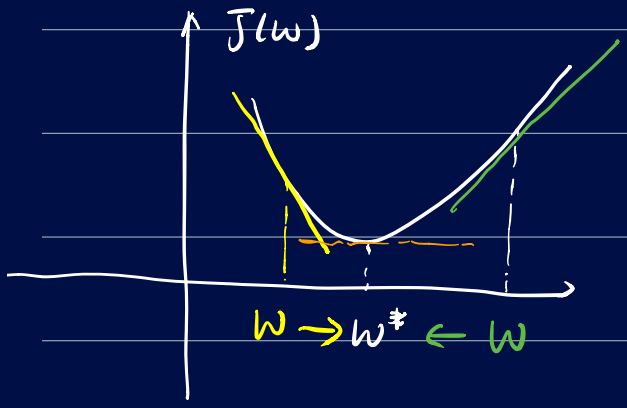
$$J(w) = \frac{1}{N}\left(\sum_{i=1}^{N} y_i^2 - \left(2\sum_{i=1}^{N} x_i y_i\right)w + \left(\sum_{i=1}^{N} x_i^2\right)w^2\right)$$

$\left(\sum_{i=1}^{N} x_i^2\right) > 0$

∪ convex

∩ concave

Gradient descent



1. We start with a random w

Repeat until $J(w)$ does not decrease any more

$$\begin{cases} w_{new} = w - \alpha \nabla J(w) \\ w \leftarrow w_{new} \end{cases}$$

α: learning rate $\alpha > 0$

$\nabla J(w)$: the gradient for $J(w)$ at w

Training a model: Finding the optimal weights (model parameters) w^* such that $J(w^*)$ is the lowest possible value.

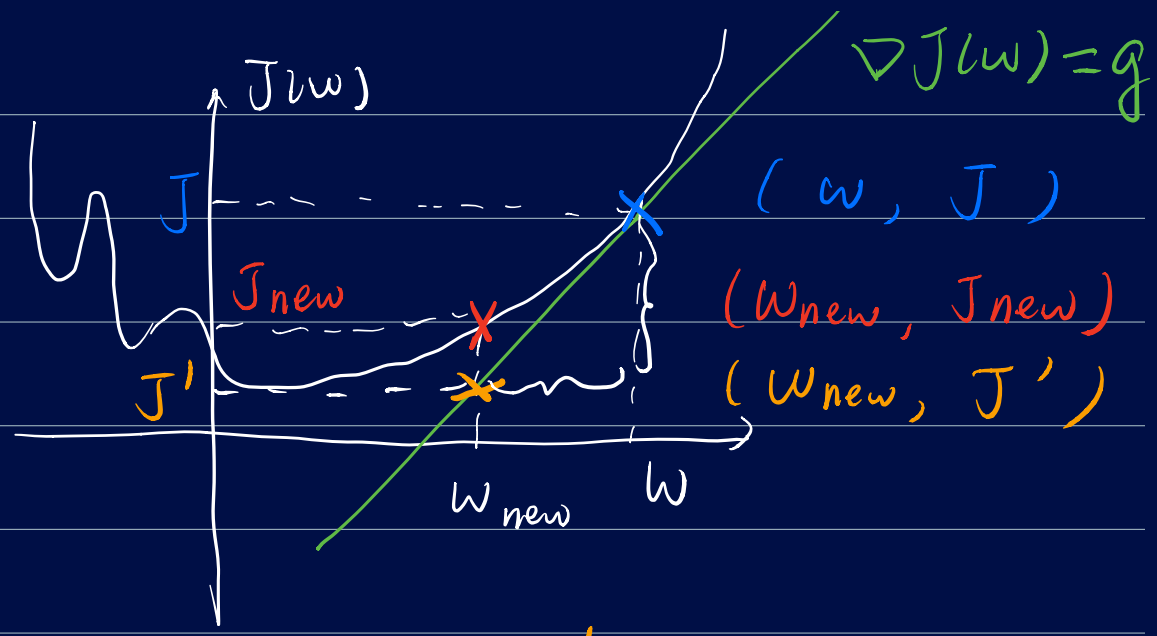
$\nabla J(w) > 0$ $\quad w_{new} = w \underbrace{- \alpha \nabla J(w)}_{<0}$

$w_{new} < w$

$\nabla J(w) < 0$ $\quad w_{new} = w \underbrace{- \alpha \nabla J(w)}_{>0}$

$w_{new} > w$

$\nabla J(w) = 0$ $\quad w_{new} = w$



$$g = \frac{J - J'}{w - w_{new}}$$

$$g(w - w_{new}) = J - J'$$

$$J' = J - g(w - w_{new})$$

$$J_{new} \approx J' = J + g(w_{new} - w)$$

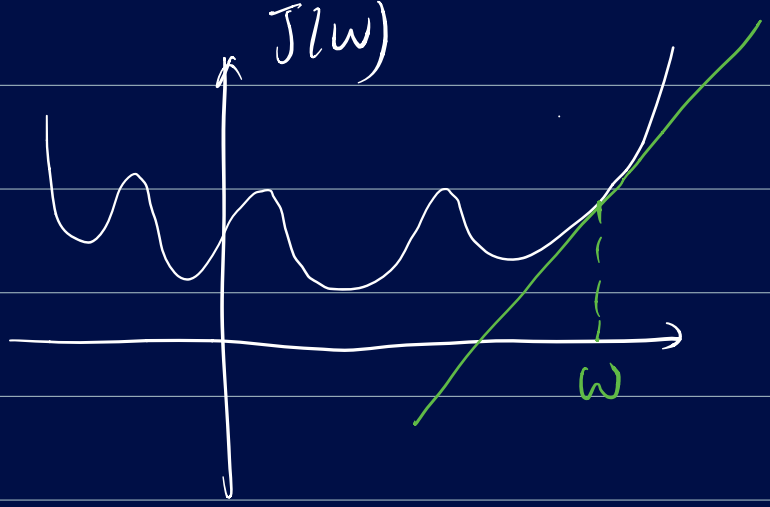
We want $J_{new} < J$ $(J_{new} - J < 0)$

$$J_{new} - J \approx g(w_{new} - w) < 0$$

$$g(w - \alpha \cdot g - w) = -\alpha g^2 < 0$$

$\alpha > 0$

Derivatives and Gradients



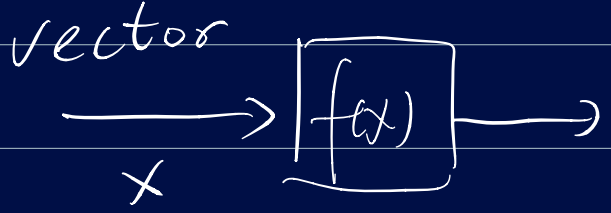
The green line: tangent line to the curve J(w)

$\nabla J(w)$: the slope of the tangent line

(∇ nabla, Δ delta)

Multivariable Case

$$f(x_1, x_2, x_3 \ldots) = f(\mathbf{x})$$



Suppose we have x and x' very close to each other $\|x - x'\| \approx 0$, then we can approximate

$$f(x) - f(x') \approx (\nabla f(x))^T (x - x')$$

$f(x) - f(x')$: Scalar

$(\nabla f(x))^T$: row vector

$(x - x')$: col. vector

$$\left.\frac{df}{dx}\right|_x = (\nabla f(x))^T$$

derivatives

Outline

Deep Learning

What does deep learning stand for?

- Deep: Neural network architectures with many hidden layers.
- Learning: Optimizing model parameters given a dataset.

In general, the deeper the model is, the more parameters we need to learn and the more data is needed.

Large-Scale Machine Learning

For deep learning systems to perform well, large datasets are required

- COCO 330K images
- ImageNet 14 million images

Challenges:

- Memory limitation: GeForce RTX 2080 Ti has 11 GB memory, while ImageNet is about 300 GB.
- Computation: Calculating gradients for the whole dataset is computationally expensive (slow), and we need to do this many times.

Stochastic Gradient Descent

Idea: Instead of calculating the gradients from the whole dataset, do it only on a subset.

- Randomly select B samples from the dataset
- The loss for this subset

$$\tilde{J}(\mathbf{w}) = \frac{1}{B}\sum_{i=1}^{B} \|y - \hat{y}_i\|^2$$

$J(w) = \frac{1}{N}\sum_{i=1}^{N} \|y - \hat{y}_i\|^2$

$B << N$

$\tilde{J}(w) = \frac{1}{B}\sum_{i=1}^{B} \|y - \hat{y}_i\|^2$

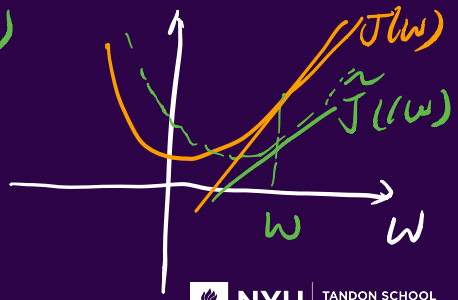
- Update Rule

  Repeat{

  $\mathbf{w}_{new} = \mathbf{w} - \alpha \nabla \tilde{J}(\mathbf{w})$

  }

$\nabla \tilde{J}(w)$



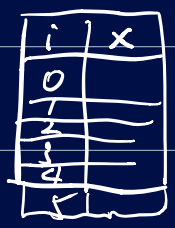
Yet Another Hyper-Parameter

This gives a noisy gradient

$$\nabla \tilde{J}(\mathbf{w}) = \nabla J(\mathbf{w}) + \epsilon$$

epsilon

- **SGD**: $B = 1$, gives very noisy gradients
- **(batch) GD**: $B = N$, $\epsilon = 0$, expensive to compute
- **Mini-batch GD**: Pick a small B, typical values are $32, 64$, rarely more than 128 for image inputs



$N = 6$

$B = 2$

B would be 16, 32, 64 ... 128

It's also a hyper-parameter

Randomly pick $B = 2$ samples from the dataset, for example

Epoch 1

iteration 1 $\tilde{J}(w) = \|y_1 - \hat{y}_1\|^2 + \|y_3 - \hat{y}_3\|^2$

iteration 2 $\hat{J}(w) = \|y_2 - \hat{y}_2\|^2 + \|y_0 - \hat{y}_0\|^2$

iteration 3 $\tilde{J}(w) = \|y_4 - \hat{y}_4\|^2 + \|y_5 - \hat{y}_5\|^2$

Epoch 2

iteration 1 $\tilde{J}(w) = \|y_4 - \hat{y}_4\|^2 + \|y_1 - \hat{y}_1\|^2$

iteration 2 $\hat{J}(w) = \|y_2 - \hat{y}_2\|^2 + \|y_3 - \hat{y}_3\|^2$

iteration 3 $\hat{J}(w) = \|y_0 - \hat{y}_0\|^2 + \|y_5 - \hat{y}_5\|^2$

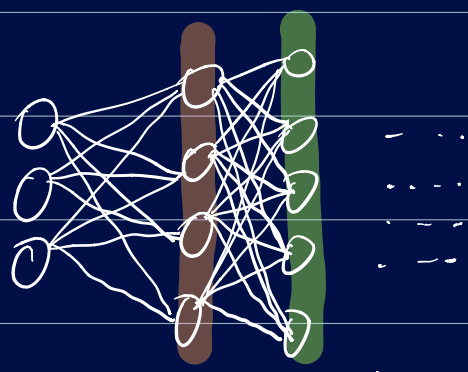
each iteration pick B samples

each epoch iterate through all N samples.

Hyper-parameters in Deep Learning

- B: batch size, # of epochs
- α: learning rate
- # of hidden layers / # neurons per layer
- Type of the activation functions for each layer

Model parameters



each edge/arrow represents a weight

all the weights / bias

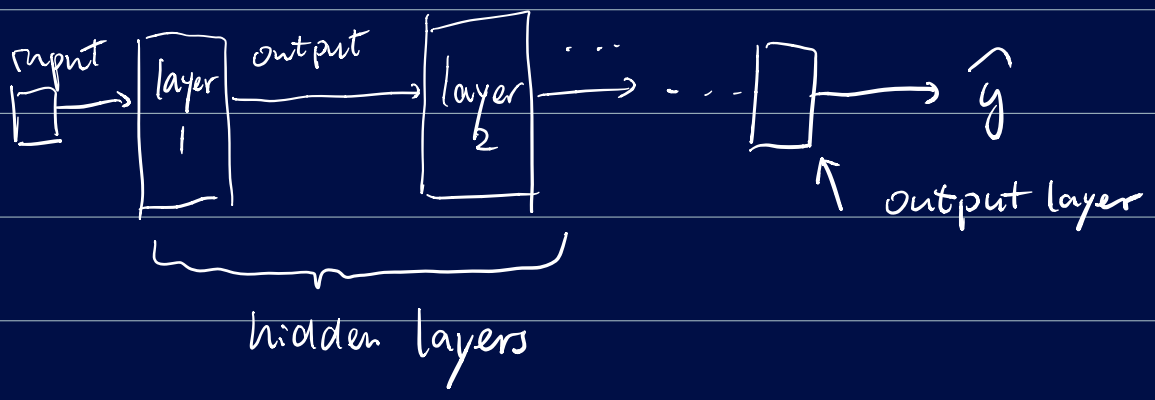
Training: to find w^* such that $J(w^*)$ is the least possible value (eg. via SGD)

$$\begin{bmatrix} w_{00} & w_{01} & w_{02} \\ w_{10} & w_{11} & w_{12} \\ w_{20} & w_{21} & w_{22} \end{bmatrix} \begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix}$$

$$= \underbrace{\begin{bmatrix} w_{01} & w_{02} \\ w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix}}_{\text{weights}} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \underbrace{\begin{bmatrix} w_{00} \\ w_{10} \\ w_{20} \end{bmatrix}}_{\text{bias}}$$

if your intial w is very close to w^*, the you will converge faster

Sometimes, it might be helpful to simply restart training with a different initialization of w (model.compile())



When you define a layer in tensorflow

Dense(10, activation="relu", input_shape=(2,))

→ creating all the weights and bias

input (B, 2) output = weights · input

ouput (B, 10) + bias

Some Noise Helps

If $\nabla \tilde{J}(w) = 0$ accidentally, at the next iteration $\nabla \tilde{J}(w)$ will be different

What would w_{new} be if $\nabla J(w) = 0$

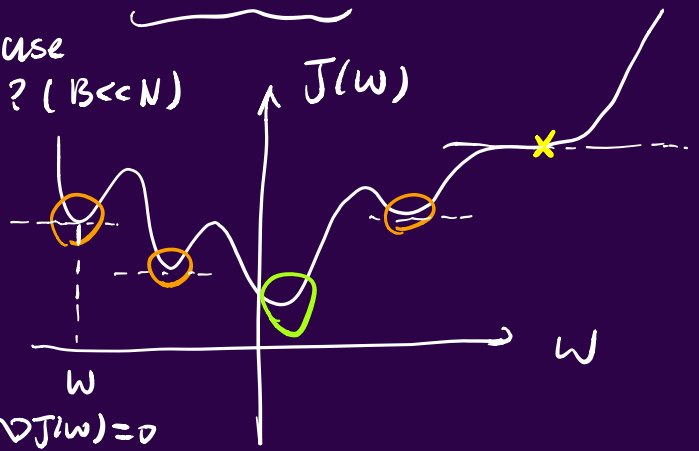
GD $w_{new} = w - \alpha \nabla J(w)$

$= w$

Even if we can, we rarely set $B = N$. In fact, some noise ($|\varepsilon| > 0$) in the gradients might help to

What if we use SGD? ($B \ll N$)

- escape from local minima,
- escape from saddle points, and
- improve generalization.



global minimum

(All global minima are also local minima)

Adam

$\nabla \tilde{J}(w) \neq \nabla J(w)$

$w_{new} = w - \alpha \underbrace{\tilde{\nabla} J(w)}_{\neq 0}$ → escape

Outline

Overparameterized Models

- Modern deep learning models are heavily overparameterized, i.e. the number of learnable parameters is much larger than the number of the training samples.

parameters per layer = # weights + # bias

Ex. input size 300 # weights 300 × 300

output size 300 # bias 300

$B = 1$

Output = Weights · input + bias (it does not depend on B)

vector of size 300 — matrix — vector of size 300

Overparameterized Models

- Modern deep learning models are heavily overparameterized, i.e. the number of learnable parameters is much larger than the number of the training samples.
  - ResNet: State-of-the-art vision model, 10-60 million parameters

Overparameterized Models

- Modern deep learning models are heavily overparameterized, i.e. the number of learnable parameters is much larger than the number of the training samples.
  - ResNet: State-of-the-art vision model, 10-60 million parameters
  - GPT-3: State-of-the-art language model, 175 billion parameters

Overparameterized Models

- Modern deep learning models are heavily overparameterized, i.e. the number of learnable parameters is much larger than the number of the training samples.
  - ResNet: State-of-the-art vision model, 10-60 million parameters
  - GPT-3: State-of-the-art language model, 175 billion parameters
- Conventional wisdom: Such models overfit.

Overparameterized Models

- Modern deep learning models are heavily overparameterized, i.e. the number of learnable parameters is much larger than the number of the training samples.
  - ResNet: State-of-the-art vision model, 10-60 million parameters
  - GPT-3: State-of-the-art language model, 175 billion parameters
- Conventional wisdom: Such models overfit.
- It is not the case in practice!

Double Descent Curve

If possible, always use wider / deeper models

Source: OpenAI

Review: Stochastic Gradient descent

Update rule: $w_{new} = w - \alpha \nabla \tilde{J}(w)$ (SGD)

Update rule for gradient descent?

$$w_{new} = w - \alpha \nabla J(w) \quad (GD)$$

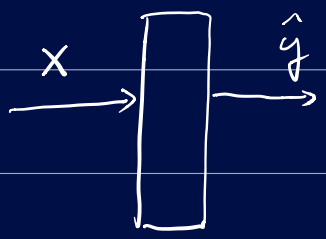
$$\tilde{J}(w) = \frac{1}{B} \sum_{i=1}^{B} \| y_i - \hat{y}_i \|^2 \quad (MSE)$$

$$\left(\frac{1}{B} \sum_{i=1}^{B} \ell(y_i, \hat{y}_i) \right) \quad \ell(y, \hat{y}) \text{ is any function}$$

that can measure the "difference" between y and $\hat{y}$

- How does batch computing work? How does model.predict() behave when we give a batch of input

$X: (B, D_{in})$ $\hat{Y} = (B, D_{out})$ $\hat{y}_i: (D_{out}, 1)$



Let's denote each row of X as x^T

$$X = \begin{bmatrix} x_1^T \\ x_2^T \\ \vdots \\ x_B^T \end{bmatrix} \qquad \hat{Y} = \begin{bmatrix} \hat{y}_1^T \\ \hat{y}_2^T \\ \vdots \\ \hat{y}_B^T \end{bmatrix}$$

the neural net is

$\hat{y} = g(Wx + b)$

$x: (D_{in}, 1)$

$\hat{y}: (D_{out}, 1)$

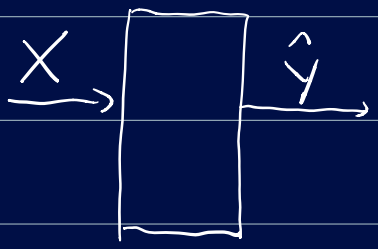
$g(z)$, z should have the same shape

$\hat{y}$ should have the same shape as $Wx + b$

$W: (D_{out}, D_{in})$

$b: (D_{out}, 1)$

model.predict(X)

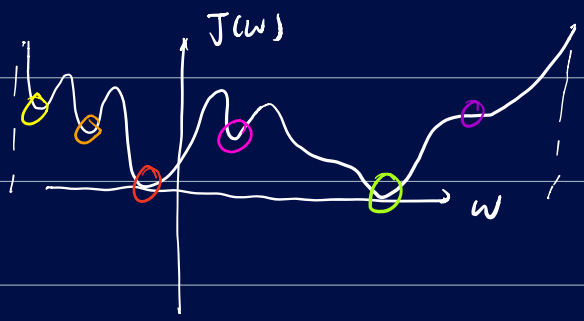


$$\hat{Y} = \begin{bmatrix} (g(Wx_1+b))^T \\ (g(Wx_2+b))^T \\ \vdots \\ (g(Wx_B+b))^T \end{bmatrix}$$

What is the # of parameters of this layer?

(parameters are contained in W and b) $D_{out} D_{in} + D_{out}$

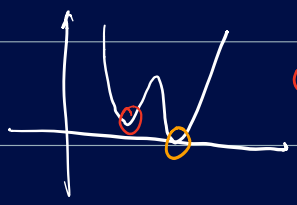
Local minima, global minima, Saddle points



Global minima

Local minima

Saddle point



$0 \approx 0$

Sometimes a local minimum is good enough

If you use MSE, then w^* must be a global minimum if $J(w^*) = 0$, $\nabla J(w^*) = 0$

$$J(w) = \frac{1}{N}\sum_{i=1}^{N} \|y_i - \hat{y}_i\|^2$$

If you choose B samples to compute $\tilde{J}(w^*)$ and $\nabla\tilde{J}(w^*)$

$J(w^*) = 0 \Rightarrow$ for all i, $\|y_i - \hat{y}_i\| = 0$ $\quad \tilde{J}(w^*) = 0$

$\Rightarrow \nabla\tilde{J}(w^*) = 0$

Outline

Let's solve the mini-project with MLPs!

Open `lab_mlp_fish_market_keras.ipynb`