# CS2109s Tutorial 7

by Lee Zong Xun

# Recap

So far, we have covered the following topics:

- Linear Regression
- Logistic Regression
- Support Vector Machines

These are algorithms to solve classification problems.

**What is one thing common to all of them?** They are all linear classifiers!

# Recap

In lecture, the perceptron model is introduced. It takes an input, aggregates it (**weighted sum**) and returns 1 only if the aggregated sum is more than some threshold else returns 0.

**Algorithm:** Perceptron Learning Algorithm

$P \leftarrow inputs \quad with \quad label \quad 1;$
$N \leftarrow inputs \quad with \quad label \quad 0;$
Initialize $\mathbf{w}$ randomly;
**while** !*convergence* **do**
  Pick random $\mathbf{x} \in P \cup N$ ;
  **if** $\mathbf{x} \in P \quad and \quad \mathbf{w}.\mathbf{x} < 0$ **then**
    $\mathbf{w} = \mathbf{w} + \mathbf{x}$ ;
  **end**
  **if** $\mathbf{x} \in N \quad and \quad \mathbf{w}.\mathbf{x} \geq 0$ **then**
    $\mathbf{w} = \mathbf{w} - \mathbf{x}$ ;
  **end**
**end**
//the algorithm converges when all the inputs are classified correctly

# Question

You must have seen multiple times now, that you need to intialize your initial weights to some random values. Why is that?

# Question

You must have seen multiple times now, that you need to intialize your initial weights to some random values. Why is that?

**The main intuition is that it allows the model to start off from random locations on the surface and therefore, reducing the chance of stagnating at some local minimas.**

# Why not initialize all weights to 1? or 2?

The key idea is to break "symmetry".

Suppose two hidden units with the same activation function are connected to the same inputs, what happens if they have the same initial parameters?

**A deterministic learning algorithm applied to a deterministic cost and model will constantly update both of these units in the same way!**

- Multiple redundant units will have the same output

# Why does the update rule work?

We have already established that when x belongs to P, we want w.x > 0, basic perceptron rule. What we also mean by that is that when x belongs to P, the angle between w and x should be _ than 90 degrees.

**Is it less than or more than 90 degrees?**

# Recall

$$cos\alpha = \frac{\mathbf{w}^T\mathbf{x}}{||\mathbf{w}||||\mathbf{x}||} \quad \Big| \quad cos\alpha \propto \mathbf{w}^T\mathbf{x}$$

$$\text{So if } \mathbf{w}^T\mathbf{x} > 0 \Rightarrow cos\alpha > 0 \Rightarrow \alpha < 90$$

$$\text{Similarly, if } \mathbf{w}^T\mathbf{x} < 0 \Rightarrow cos\alpha < 0 \Rightarrow \alpha > 90$$

# How does it work 🤯?

$(\alpha_{new})$ when $\mathbf{w_{new}} = \mathbf{w} + \mathbf{x}$

$$
\begin{aligned}
cos(\alpha_{new}) &\propto \mathbf{w_{new}}^T \mathbf{x} \\
&\propto (\mathbf{w} + \mathbf{x})^T \mathbf{x} \\
&\propto \mathbf{w}^T \mathbf{x} + \mathbf{x}^T \mathbf{x} \\
&\propto cos\alpha + \mathbf{x}^T \mathbf{x} \\
cos(\alpha_{new}) &> cos\alpha
\end{aligned}
$$

$(\alpha_{new})$ when $\mathbf{w_{new}} = \mathbf{w} - \mathbf{x}$

$$
\begin{aligned}
cos(\alpha_{new}) &\propto \mathbf{w_{new}}^T \mathbf{x} \\
&\propto (\mathbf{w} - \mathbf{x})^T \mathbf{x} \\
&\propto \mathbf{w}^T \mathbf{x} - \mathbf{x}^T \mathbf{x} \\
&\propto cos\alpha - \mathbf{x}^T \mathbf{x} \\
cos(\alpha_{new}) &< cos\alpha
\end{aligned}
$$

# Optional: Why don't we use linear regression for classification tasks?

# Question 1

**Determine a function that can be used to model the decision boundaries of the logical NOT, XOR, and AND gates**. What are the weights and bias for each perceptron respectively? The steps are given below:

- Initialise all your weights to 0 (including bias term)
- Positive class is 1 and negative class is 0
- Use the Perceptron Update Rule and the learning rate from Lecture 8 ($\eta = 0.1$)
- Update the model for each misclassified instance in the \textbf{given fixed order} of data samples
- Activation function is the modified $\mathrm{sign}(z) = 1 \text{ if } \mathrm{z} \ \geq 0 \text{ else } 0$

# Question 1

How should we approach this question?

| AND | | |
|---|---|---|
| x1 | x2 | y |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

| OR | | |
|---|---|---|
| x1 | x2 | y |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

| NOT | |
|---|---|
| x | y |
| 0 | 1 |
| 1 | 0 |

# Question 1

```
def perceptron(x, y, w=None, lr=0.1, iters=10):
  """Return perceptron model weights trained by Perceptron Update:
  w <- w + lr * (y - sign(w * x)) * x
  """
  if w is None:
      w = np.zeros([1, len(x[0]) + 1]) # + 1 for bias
  x = np.concatenate((np.ones([len(x), 1]), x), axis=1) # concat 1 for bias
  step = [w]
  for iter in range(iters):
      for i in range(len(x)):
          out = np.inner(w, x[i])
          label = sign_fn(out)

          if label != y[i]:
              w = w + lr * (y[i] - label) * x[i]
  return w
```

# Question 1

Using the Perceptron Update Rule, we will get the following weights:

$$\text{AND Gate: } w_1 = 0.1, w_2 = 0.2, b = -0.3$$
$$\text{OR Gate: } w_1 = 0.1, w_2 = 0.1, b = -0.1$$
$$\text{NOT Gate: } w_1 = -0.1, b = 0.$$

# Question 1

**Is it possible to model the XOR function using a single Perceptron?**

**Solution:** No! Requires an intermediate hidden layer for preliminary transformation in order to achieve the logic of an XOR gate.

# Question 1

Model XOR function (takes $2$ inputs) using a number of perceptrons that implement AND, OR, and NOT functions. Show the diagram of the final Perceptron network. What can you conclude now that you have intermediate functions acting on the inputs and not just a single perceptron unit?

**Solution:** $XOR(x_1, x_2) = AND(NOT(AND(x_1, x_2)), OR(x_1, x_2))$. We resort to using these intermediate "layers" that help with extra computation that is not possible with a single perceptron. This shows us how stacking perceptrons enables us to model complex behaviors in data.

# Question 1

If we change the ordering of data samples in Perceptron Update Rule, will the model converges to a different model weight for the AND operator? What can you conclude from the observation? (Hint: Use your solution from (a), and try switching the row-orderings of the input data from $(0, 1, 2, 3)$ to $(0, 2, 3, 1)$ and $(0, 2, 1, 3)$ respectively)

**Solution**: For ordering $(0, 2, 3, 1)$, the weight differs from the original ordering after two steps and takes more steps to converge to the same optimum model $(-0.2, 0.1, 0.2)$ at the end.

For ordering $(0, 2, 1, 3)$, it follows the same initial update steps as the original ordering but converges to a optimum model $(-0.2, 0.2, 0.1)$.

# Question 1

From the above observations, we can conclude that:

- Reordering data points could help the model converge much faster.
- Manipulating the ordering could direct the model to a different weight. There is no guarantee to converge to the same model even for such a simple gate function.

# Question 1

Does your proposed model have high bias and high variance?

**Solution:** The proposed model has low bias and low variance. All these models perform as expected to the AND gate, and the model is powerful enough to learn the function. The linear model is the simplest model to perform the AND gate, and these different models are similar to each others.

# Question 2

Suppose you have a dataset of 500 product sales, each with three input features (TV, radio, and newspaper advertising expenditure) and a continuous label indicating the sales.

You decide to use a single-layer perceptron and a multi-layer perceptron to make predictions. After training both networks, you obtain a mean squared error of 1000 on the training set and a mean squared error of 2000 on the validation set for the single-layer perceptron, and a mean squared error of 800 on the training set and a mean squared error of 1200 on the validation set for the multi-layer perceptron.

**What might be the reasons for the difference in performance between the single-layer perceptron and the multi-layer perceptron?**

**Solution:**

- single-layer perceptron is a linear classifier,
- multi-layer perceptron can learn non-linear decision boundaries.

In this case, the multi-layer perceptron is able to capture more complex relationships between the input features and the output label, leading to better performance on the validation set.

**How might you modify the single-layer perceptron to improve its performance, and what are the advantages and disadvantages of doing so?**

**Solution:**

- Add more features, such as polynomial or interaction terms, to capture non-linear relationships between the input features and the output label.
- But, may be computationally expensive and lead to overfitting if the number of features is too high.

**What techniques could you use to improve the performance of the multi-layer perceptron?**

**Solution:**

- Use a different activation function, such as a sigmoid or a hyperbolic tangent, which can learn more complex decision boundaries than the step function used by the single-layer perceptron.
- Add more hidden layers to the multi-layer perceptron to increase its capacity to learn more complex patterns, but this approach can also increase the risk of overfitting.

*Fun fact*: According to the *Universal Approximation Theorem*, one hidden layer is actually sufficient to model any function 🤔

Use **regularization techniques**: Dropout is a regularization technique that randomly drops out a percentage of neurons during training to prevent co-adaptation.

Use **batch normalization**: Batch normalization can improve the performance of an MLP by normalizing the inputs to each layer, which helps to reduce the effects of covariate shift.

Use different **optimization** algorithms: The choice of optimization algorithm can also affect the performance of an MLP. Gradient descent is the most commonly used optimization algorithm, but there are many other algorithms such as Adam, RMSprop, and Adagrad that may perform better for certain problems.

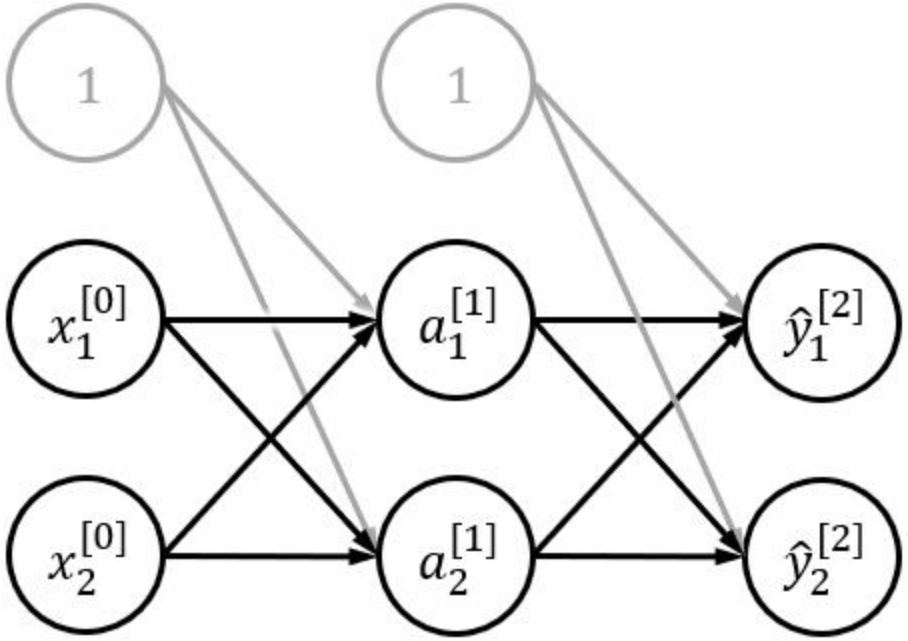
Use **ensemble** methods: Ensemble methods such as bagging and boosting can improve the performance of an MLP by combining multiple models trained on different subsets of the data. This can help to reduce variance and improve generalization.

Use **early stopping**: Early stopping can prevent overfitting by stopping the training process when the validation loss stops improving. This can help to prevent the model from memorizing the training data and improve its ability to generalize to new data.

# Question 3

In this question, we are going to use a neural network with **a 2-D input, one hidden layer with two neurons, and two output neurons**. Additionally, the hidden neurons and the input will **include a bias**. We use **ReLU function** as the non-linear activation function. FYI: $\mathrm{ReLU}(x) = \max(0, x)$.

Suppose there is a data input $\mathbf{x} = (2, 3)^\top$ and the actual output label is $\mathbf{y} = (0.1, 0.9)^\top$. The weights for the network are




$$\boldsymbol{W}^{[1]} = \begin{bmatrix} 0.1 & 0.1 \\ -0.1 & 0.2 \\ 0.3 & -0.4 \end{bmatrix}, \boldsymbol{W}^{[2]} = \begin{bmatrix} 0.1 & 0.1 \\ 0.5 & -0.6 \\ 0.7 & -0.8 \end{bmatrix},$$

Of course, we also include biases of value $b = 1$ for both hidden and output layers. Calculate the following values after the forward propagation: $\mathbf{a}^{[1]}$, $\hat{\mathbf{y}}^{[2]}$ and $L(\hat{\mathbf{y}}^{[2]}, \mathbf{y})$. Here, we use MSE (mean squared error) loss function.

$$\mathbf{a}^{[1]} = ReLU((\mathbf{W}^{[1]})^\top \mathbf{X})$$

$$\begin{aligned} a_1^{[1]} &= ReLU(x_0 \times W_{01}^{[1]} + x*1 \times W_{11}^{[1]} + x*2 \times W_{21}^{[1]}) \\ &= ReLU(0.1 + 2 \times (-0.1) + 3 \times 0.3) \\ &= ReLU(0.8) \\ &= 0.8 \end{aligned}$$

$$\begin{aligned} a_2^{[1]} &= ReLU(x*0 \times W_{02}^{[1]} + x*1 \times W_{12}^{[1]} + x*2 \times W_{22}^{[1]}) \\ &= ReLU(0.1 + 2 \times 0.2 + 3 \times (-0.4)) \\ &= ReLU(-0.7) \\ &= 0 \end{aligned}$$

$$\hat{\mathbf{y}}^{[2]} = ReLU((\mathbf{W}^{[2]})^\top \mathbf{a}^{[1]})$$

$$\begin{aligned}
\hat{y}^{[2]} * 1 &= ReLU(a_0^{[1]} \times W_{01}^{[2]} + a_1^{[1]} \times W_{11}^{[2]} + a_2^{[1]} \times W_{21}^{[2]}) \\
&= ReLU(0.1 + 0.8 \times 0.5 + 0 \times 0.7) \\
&= ReLU(0.5) \\
&= 0.5
\end{aligned}$$

$$\begin{aligned}
\hat{y}^{[2]} * 2 &= ReLU(a_1^{[0]} \times W_{02}^{[2]} + a_1^{[1]} \times W_{12}^{[2]} + a_2^{[1]} \times W_{22}^{[2]}) \\
&= ReLU(0.1 + 0.8 \times (-0.6) + 0 \times (-0.8)) \\
&= ReLU(-0.38) \\
&= 0
\end{aligned}$$

$$\begin{aligned}
L(\hat{\mathbf{y}}^{[2]}, \mathbf{y}) &= \frac{1}{2} \times ((\hat{y}^{[2]}_1 - y_1)^2 + (\hat{y}^{[2]}_2 - y_2)^2) \\
&= \frac{1}{2} \times ((0.5 - 0.1)^2 + (0 - 0.9)^2) \\
&= \frac{1}{2} \times (0.16 + 0.81) \\
&= 0.485
\end{aligned}$$

# Question 4

We can define a neural network as follows:

$$\hat{y} = g(\mathbf{W}^{[\mathbf{L}]^\mathbf{T}} \dots g(\mathbf{W}^{[\mathbf{2}]^\mathbf{T}} \cdot g(\mathbf{W}^{[\mathbf{1}]^\mathbf{T}} x)))$$

where $\mathbf{W}^{[\mathbf{l}] \in \{\mathbf{1}, \cdots, \mathbf{L}\}}$ is a weight matrix. You're given the following weight matrices:

$$\mathbf{W}^{[\mathbf{3}]} = \begin{bmatrix} 1.2 & -2.2 \\ 1.2 & 1.3 \end{bmatrix}, \mathbf{W}^{[\mathbf{2}]} = \begin{bmatrix} 2.1 & -0.5 \\ 0.7 & 1.9 \end{bmatrix}, \mathbf{W}^{[\mathbf{1}]} = \begin{bmatrix} 1.4 & 0.6 \\ 0.8 & 0.6 \end{bmatrix}$$

Furthermore, you are given $g(z) = \mathrm{SiLU}(z) = \frac{z}{1+e^{-z}}$ between all layers *except the last layer*.

Is it possible to replace the whole neural network with just one matrix in both cases **with** and **without** non-linear activations $g(z)$? For both cases, either shows that it is possible by providing such a matrix or prove that there exists no such matrix. What does this signify about the importance of the non-linear activation?

**Solution:** Without $\mathrm{SiLU}(z)$, we can show that we can replace the neural network with the following matrix $\mathbf{M^T}$:

$$
\begin{aligned}
M^T &= \begin{bmatrix} 1.2 & -2.2 \\ 1.2 & 1.3 \end{bmatrix}^T \begin{bmatrix} 2.1 & -0.5 \\ 0.7 & 1.9 \end{bmatrix}^T \begin{bmatrix} 1.4 & 0.6 \\ 0.8 & 0.6 \end{bmatrix}^T \\
&= \begin{bmatrix} 4.56 & 3.408 \\ -6.82 & -3.658 \end{bmatrix}
\end{aligned}
$$

With $\mathrm{SiLU}(z)$, assume there exists such a matrix $\mathbf{M^T}$. Consider 2 inputs,

$x_1 = [1 \quad 0]$ and $x_2 = [2 \quad 0]$:

$$\hat{y_1} = \begin{bmatrix} 1.2 & -2.2 \\ 1.2 & 1.3 \end{bmatrix}^T g\left( \begin{bmatrix} 2.1 & -0.5 \\ 0.7 & 1.9 \end{bmatrix}^T g\left( \begin{bmatrix} 1.4 & 0.6 \\ 0.8 & 0.6 \end{bmatrix}^T \begin{bmatrix} 1 \\ 0 \end{bmatrix} \right) \right)$$

$$= \begin{bmatrix} 3.0571 \\ -5.2727 \end{bmatrix}$$

$$\hat{y_2} = \begin{bmatrix} 1.2 & -2.2 \\ 1.2 & 1.3 \end{bmatrix}^T g\left( \begin{bmatrix} 2.1 & -0.5 \\ 0.7 & 1.9 \end{bmatrix}^T g\left( \begin{bmatrix} 1.4 & 0.6 \\ 0.8 & 0.6 \end{bmatrix}^T \begin{bmatrix} 2 \\ 0 \end{bmatrix} \right) \right)$$

$$= \begin{bmatrix} 7.7257 \\ -13.2458 \end{bmatrix}$$

Since $x_2 = 2x_1$, $\mathbf{M^T} x_2 = 2\mathbf{M^T} x_1 \implies \hat{y_2} = 2\hat{y_1}$ by linearity of $\mathbf{M^T}$. But by the computation we did above, $\hat{y_2} \neq 2\hat{y_1}$, thus there exist no such $\mathbf{M^T}$.

Without non-linear activations, the entire network **collapses to a simple linear model**. It's as good as using a network with one layer; we can no longer capitalise on this "depth" of the original neural network (i.e., $L$ layers deep).

$$\begin{aligned}\hat{y} &= \mathbf{W}^{[\mathbf{L}]\mathbf{T}} \dots \mathbf{W}^{[\mathbf{2}]\mathbf{T}} \mathbf{W}^{[\mathbf{1}]\mathbf{T}} x \\ &= \mathbf{A}x, \quad \text{where } \mathbf{A} = \mathbf{W}^{[\mathbf{L}]\mathbf{T}} \dots \mathbf{W}^{[\mathbf{2}]\mathbf{T}} \mathbf{W}^{[\mathbf{1}]\mathbf{T}} \text{ by matrix multiplication}\end{aligned}$$

For non-linear data, these non-linear activation functions let the network model those relationships. Without non-linear activations, all the parameters in the network behave the same way, lowering the representation power or "learning capacity" the neural network.

You're building a self-driving car program that takes in grayscale images of size $32 \times 32$ where 32 is the image height and width. There are 4 classes your simplified program has to classify: $\{\text{car}, \text{person}, \text{traffic light}, \text{stop sign}\}$. You start off experimenting with a Multi-layer Perceptron composed of three linear layers of the form $y = W^T x$, where $x \in \mathcal{R}^d$ is the input vector, $W$ is the weight matrix, and $y$ is the network output.

What are the dimensions of the input vector, the weight matrix, and the output vector of the three linear layers, given the following details? Assume the batch size is 1.

**Solution:**

| layer | Input dim | Weight Matrix dim | Output dim |
|---|---|---|---|
| Linear layer 1 | $1024 \times 1$ | $1024 \times 512$ | $512 \times 1$ |
| Linear layer 2 | $512 \times 1$ | $512 \times 128$ | $128 \times 1$ |
| Linear layer 3 | $128 \times 1$ | $128 \times 4$ | $4 \times 1$ |