Machine Learning

Neural Networks

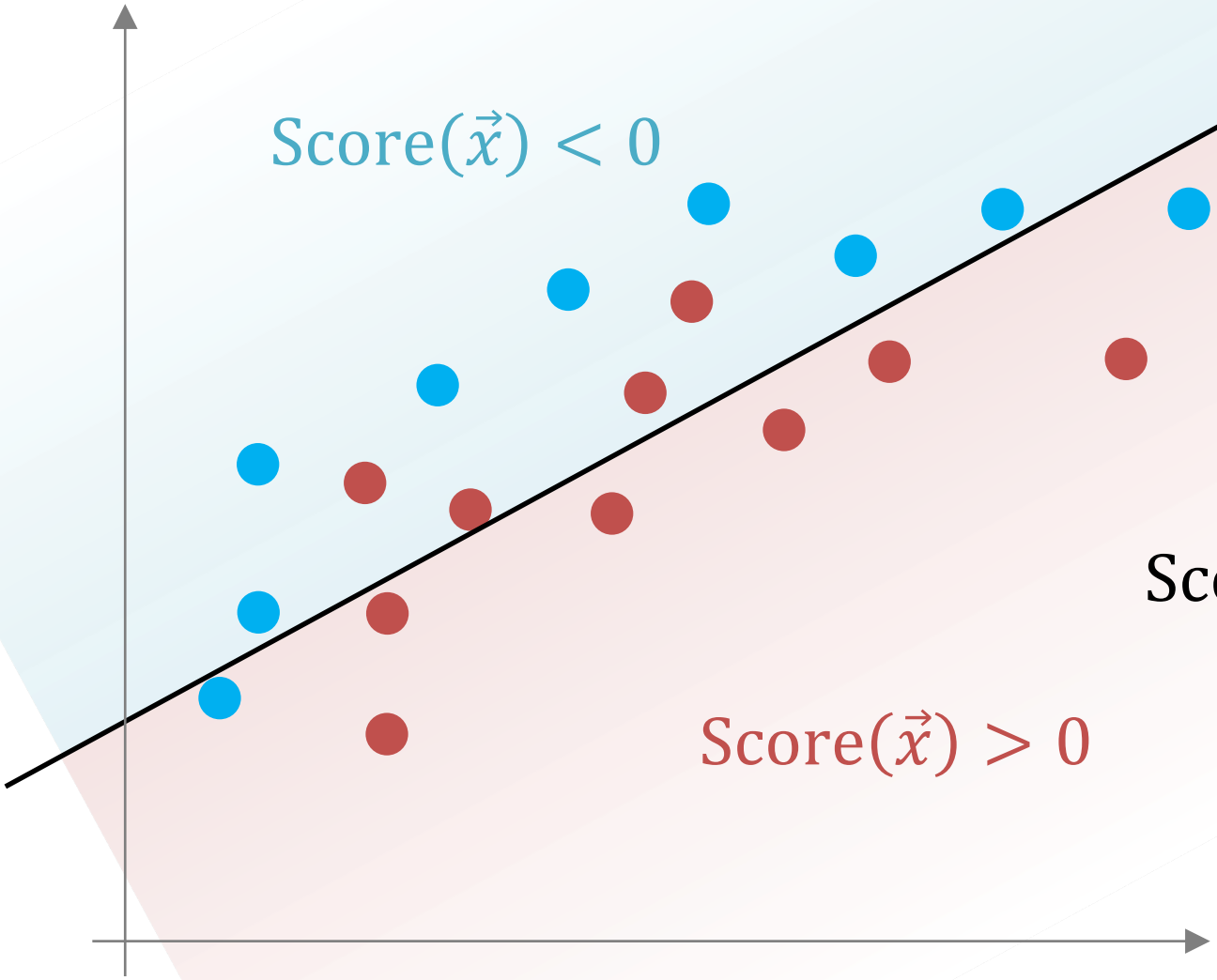
Tarapong Sreenuch

8 February 2024

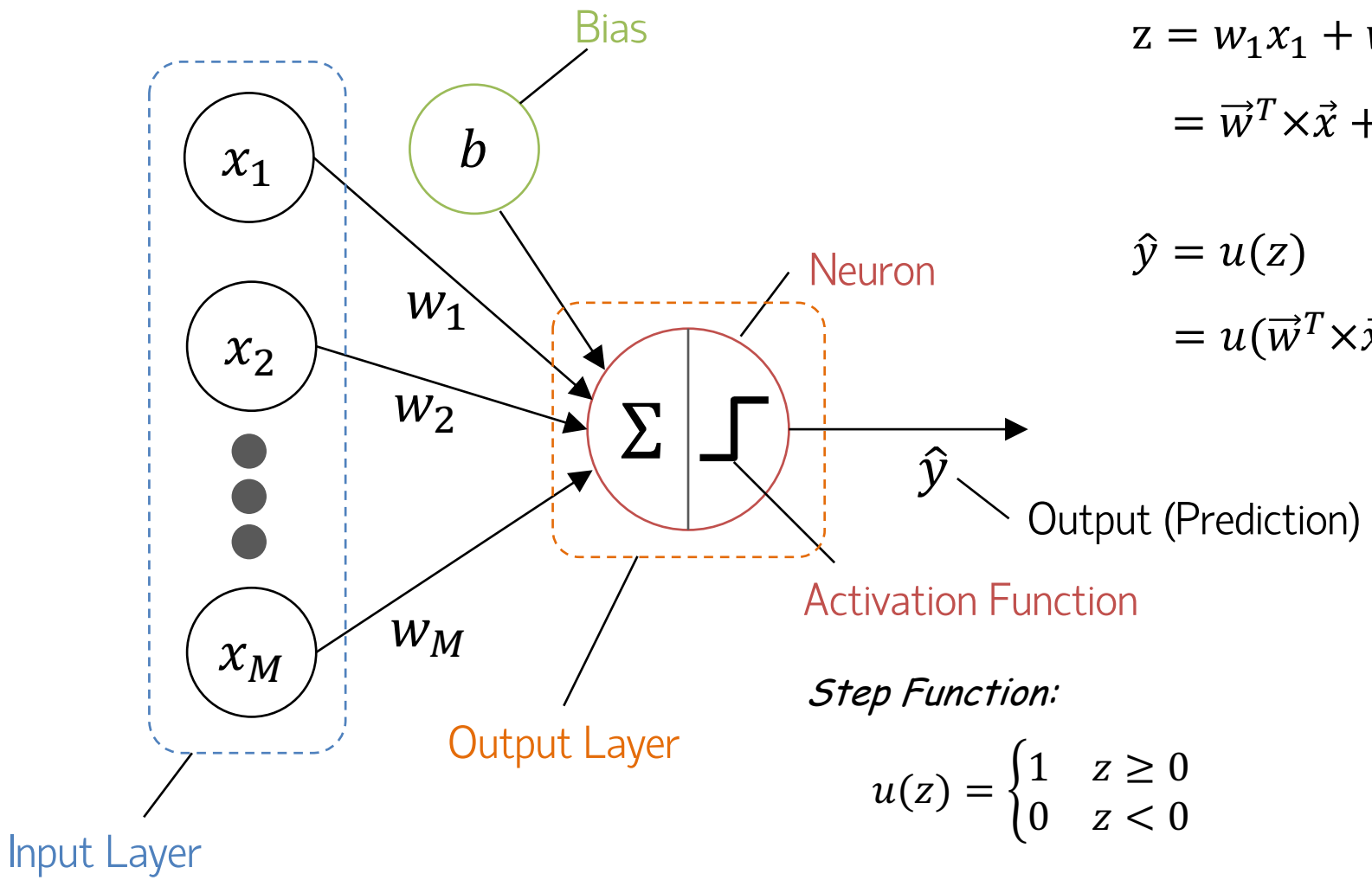
克明峻德，格物致知


Q: What are the colours we need to restore the colour of this drawing.
A: 3, minimally, e.g. red, green & blue. Any colour can be generated from these 3s.
We call them a 'basis' (in Math). In ML, we call them features.
The features are often engineered. Can they be self-learned from the data?
(Deep Learning)

Linear Classifier



Perceptron

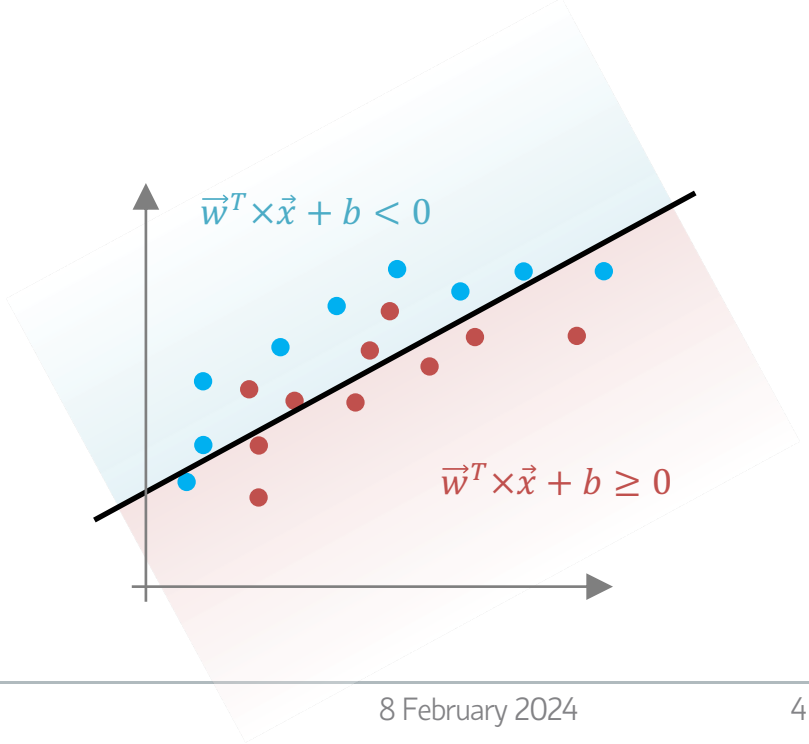


$$z = w_1x_1 + w_2x_2 + \cdots + w_Mx_M + b$$
$$= \vec{w}^T \times \vec{x} + b$$

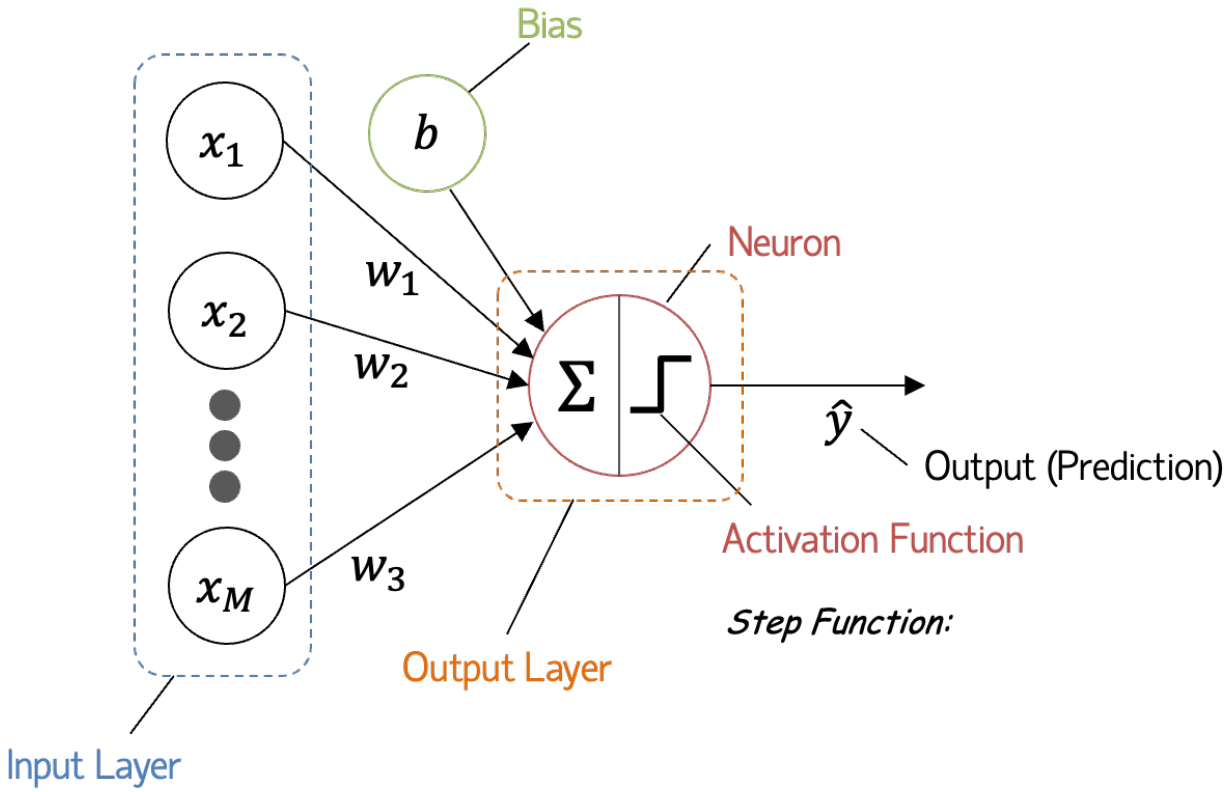
$$\hat{y} = u(z)$$
$$= u(\vec{w}^T \times \vec{x} + b) \Rightarrow \hat{y} = \begin{cases} 1 & \vec{w}^T \times \vec{x} + b \geq 0 \\ 0 & \vec{w}^T \times \vec{x} + b < 0 \end{cases}$$

Step Function:

$$u(z) = \begin{cases} 1 & z \geq 0 \\ 0 & z < 0 \end{cases}$$



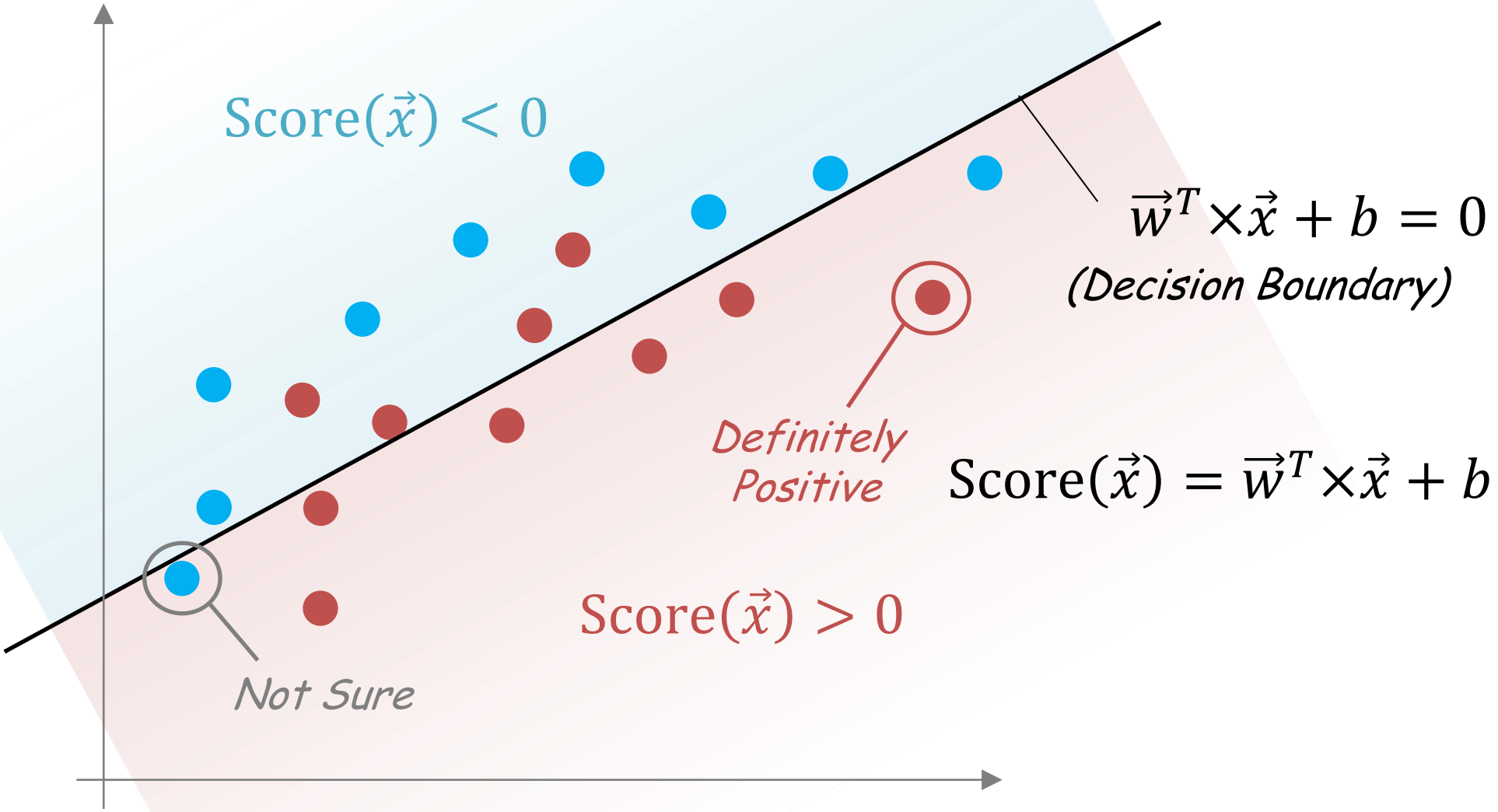
Tensorflow Code Snippet




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

num_inputs = M          # input dimension

model = Sequential([
    Dense(1,
          activation="step",
          input_shape=(num_inputs,))
])
```

TensorFlow

How Confidence?



Scoring Interpretation

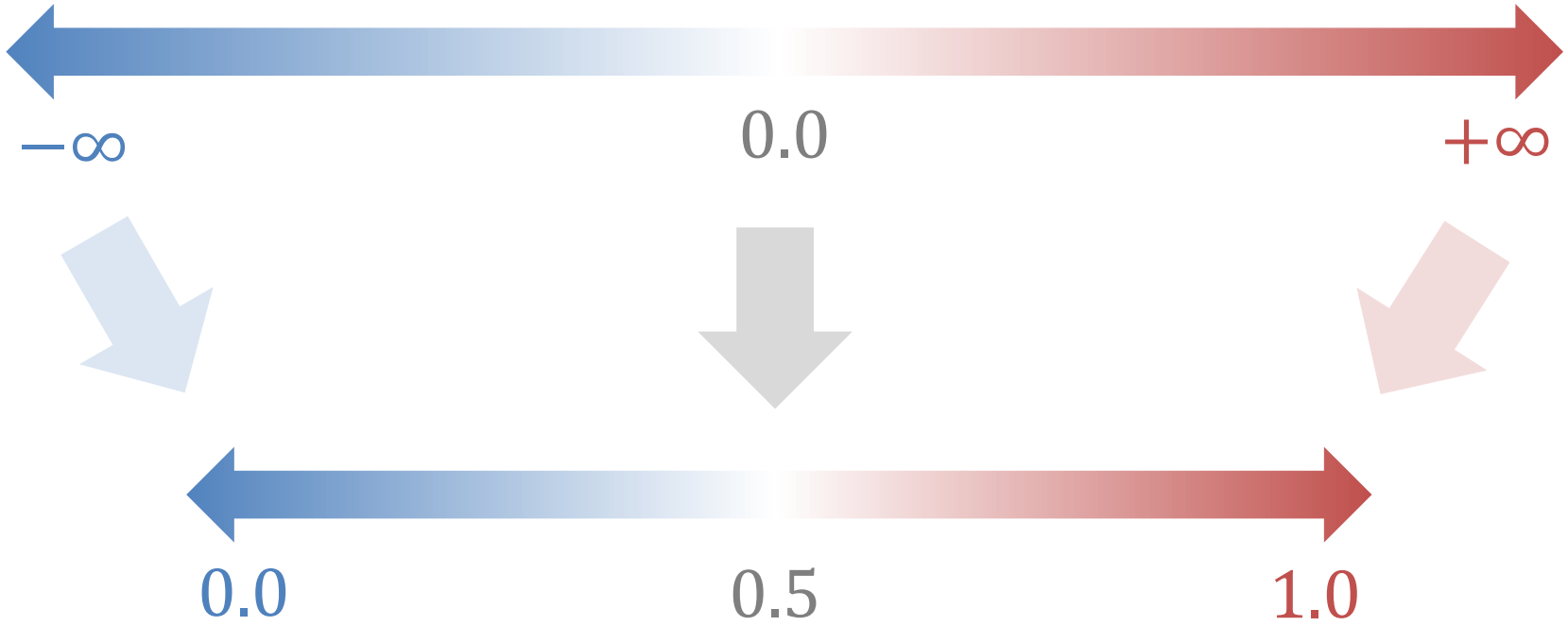
$$\text{Score}(\vec{x}^{(k)}) = \vec{w}^T \times \vec{x}^{(k)} + b$$

$-\infty$ $\quad \hat{y}^{(k)} = 0 \quad$ 0.0 $\quad \hat{y}^{(k)} = 1 \quad +\infty$

Very Sure	*Not Sure If*	*Very Sure*
$\hat{y}^{(k)} = 0$	$\hat{y}^{(k)} = 0 \text{ or } 1$	$\hat{y}^{(k)} = 1$
$P(y = 1 \mid \vec{x}^{(k)}, \vec{w}, b) = 0$	$P(y = 1 \mid \vec{x}^{(k)}, \vec{w}, b) = 0.5$	$P(y = 1 \mid \vec{x}^{(k)}, \vec{w}, b) = 1$

Perceptron Model

$$\text{Score}(\vec{x}^{(k)}) = \vec{w}^T \times \vec{x}^{(k)} + b$$

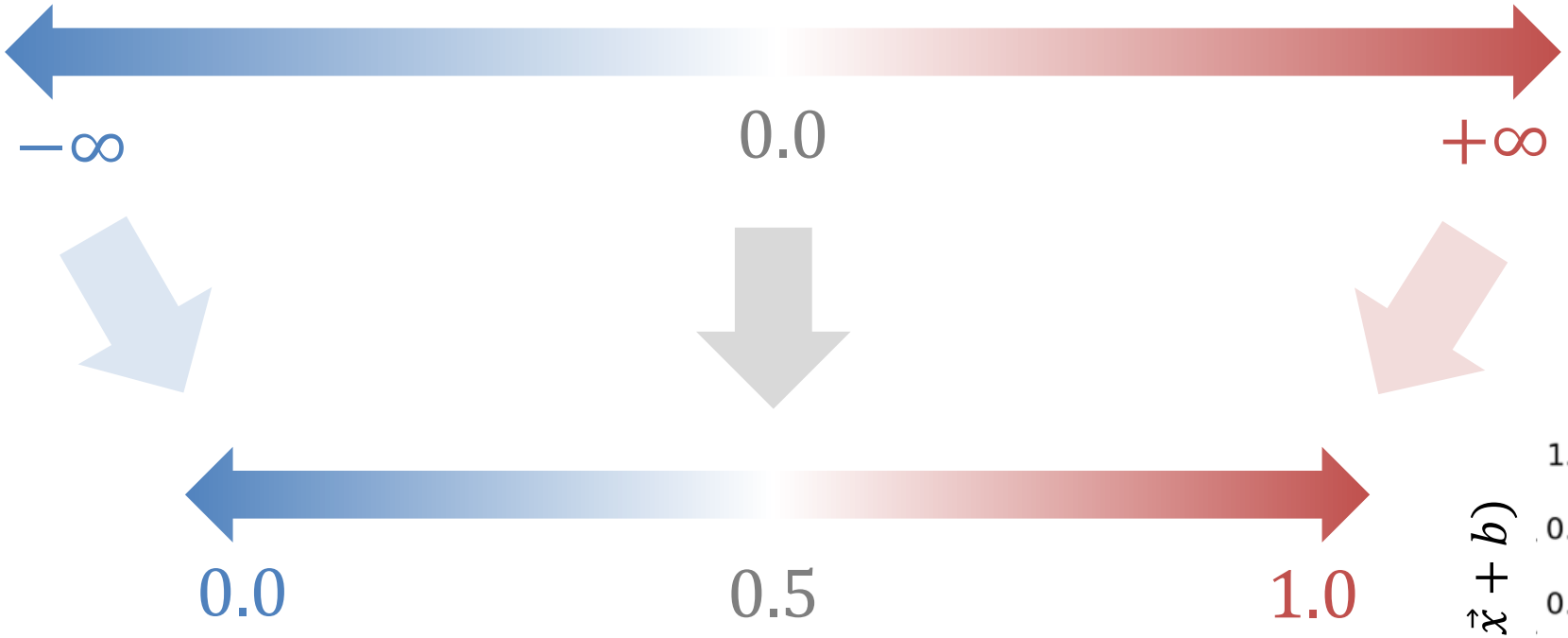


$$P\left(y = 1 \,\middle|\, \vec{x}^{(k)}\right)$$

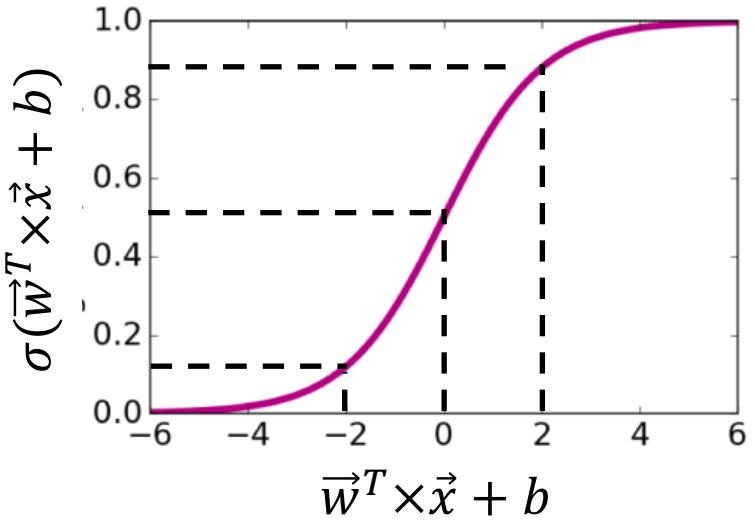
Q: How do we map (-∞, +∞) to [0.0, 1.0]?

Sigmoid (Logit) Function

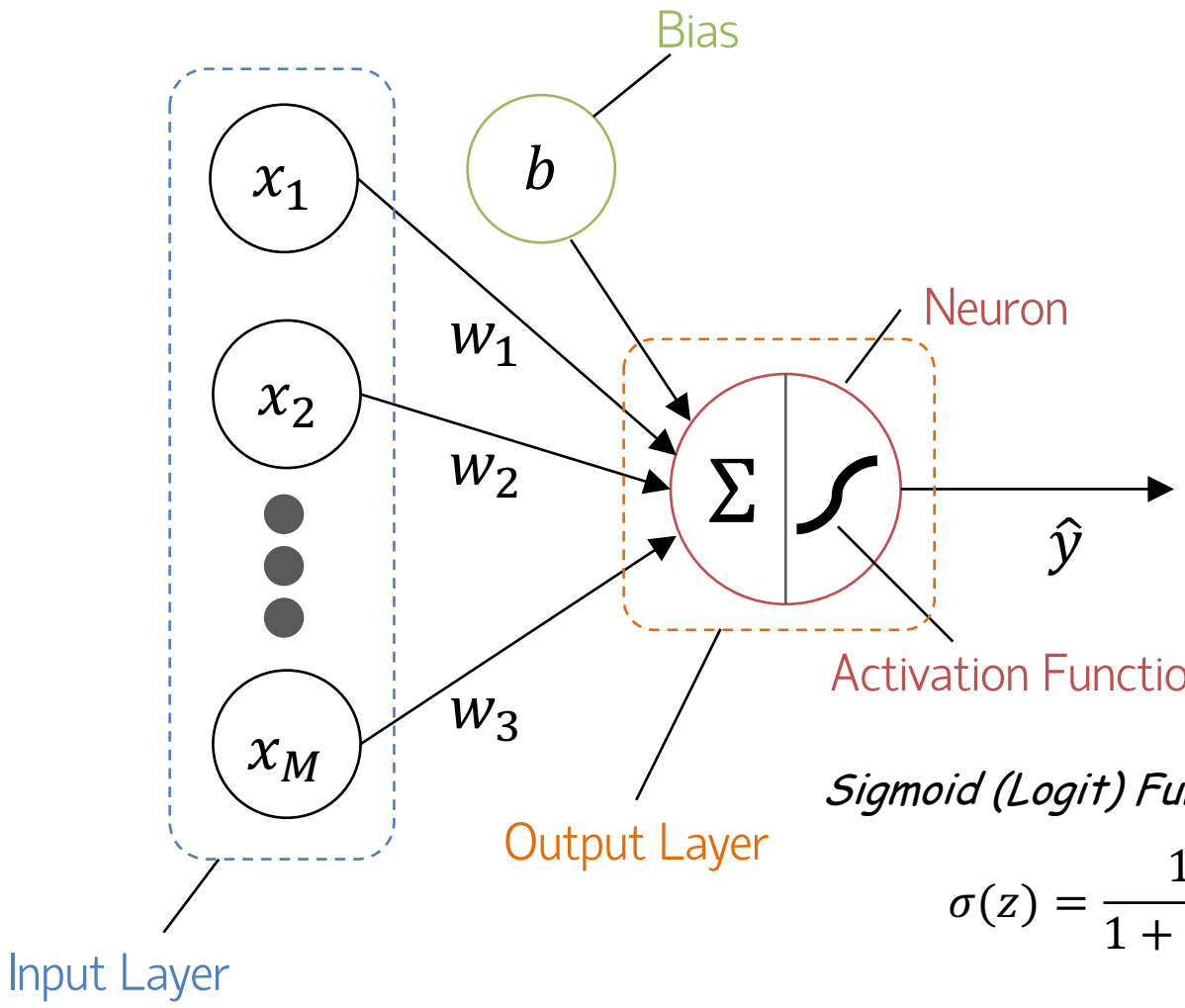
$$\text{Score}(\vec{x}^{(k)}) = \vec{w}^T \times \vec{x}^{(k)} + b$$



$$P\left(y = 1 \,\middle|\, \vec{x}^{(k)}, \vec{w}, b\right) = \frac{1}{1+e^{-(\vec{w}^T \times \vec{x}^{(k)}+b)}}$$



Perceptron



$$z = w_1 x_1 + w_2 x_2 + \cdots + w_M x_M + b$$
$$= \vec{w}^T \times \vec{x} + b$$

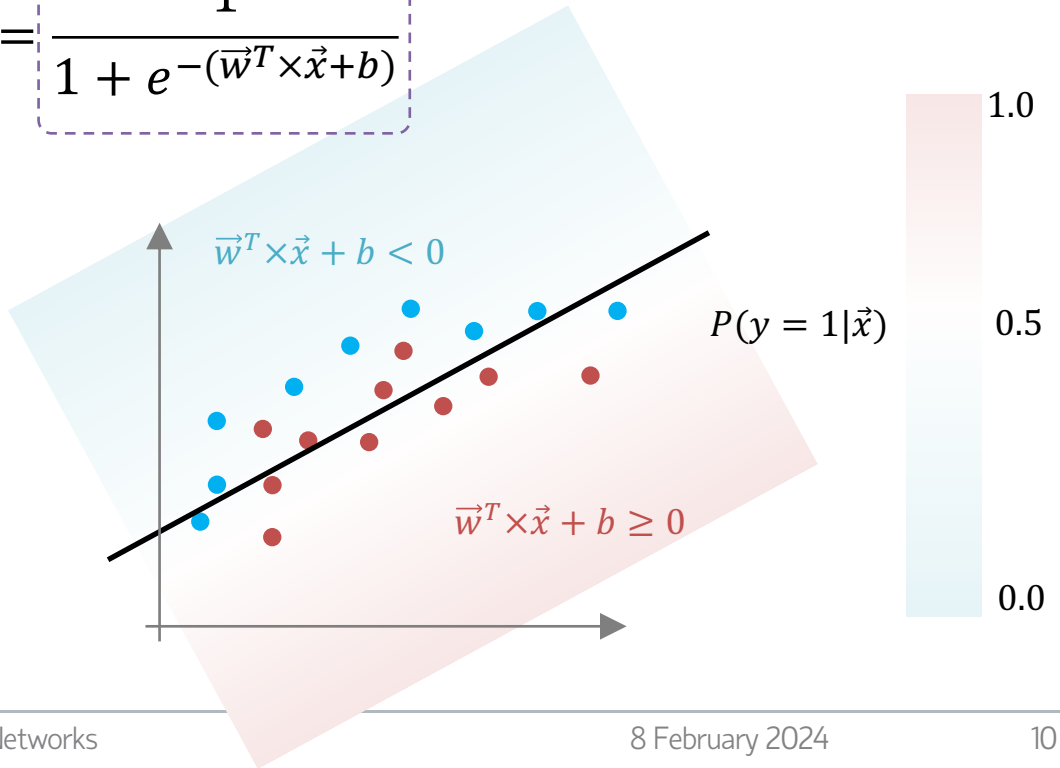
$$\hat{y} = \sigma(z)$$
$$= \sigma(\vec{w}^T \times \vec{x} + b)$$
$$= \frac{1}{1 + e^{-(\vec{w}^T \times \vec{x} + b)}}$$

$P(y = 1|\vec{x})$

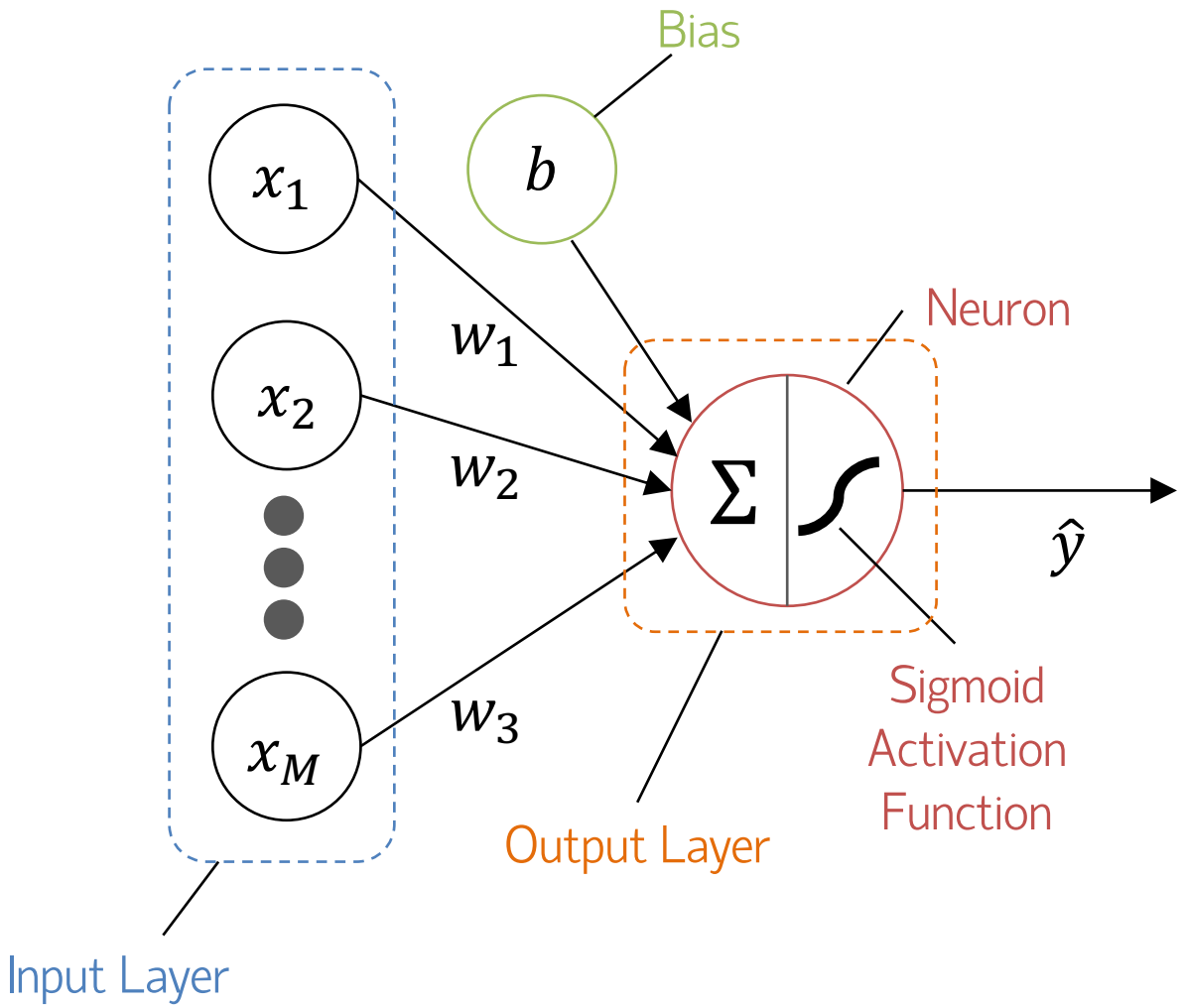
Posterior Probability

Sigmoid (Logit) Function:

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



Tensorflow Code Snippet



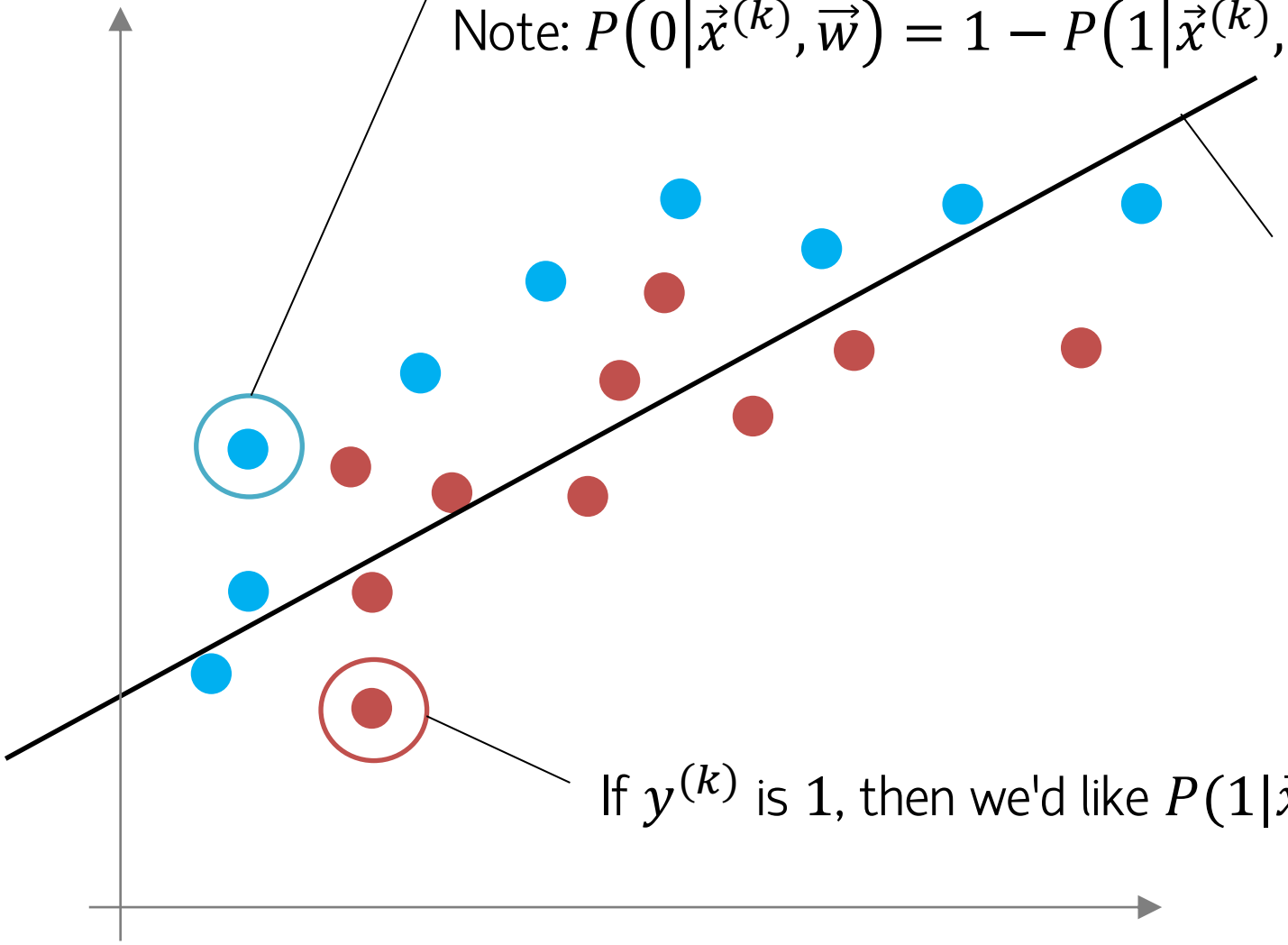

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense

num_inputs = M          # input dimension
num_outputs = 1         # output dimension

model = Sequential([
    Dense(num_outputs,
          activation="sigmoid”,
          input_shape=(num_inputs,))
])
```

TensorFlow

What We Are Optimising

If $y^{(k)}$ is 0, then we'd like $P\left(0\middle|\vec{x}^{(k)},\vec{w}\right) \rightarrow 1$ (or$P\left(1\middle|\vec{x}^{(k)},\vec{w}\right) \rightarrow 0$).

Note: $P\left(0\middle|\vec{x}^{(k)},\vec{w}\right) = 1 - P\left(1\middle|\vec{x}^{(k)},\vec{w}\right)$

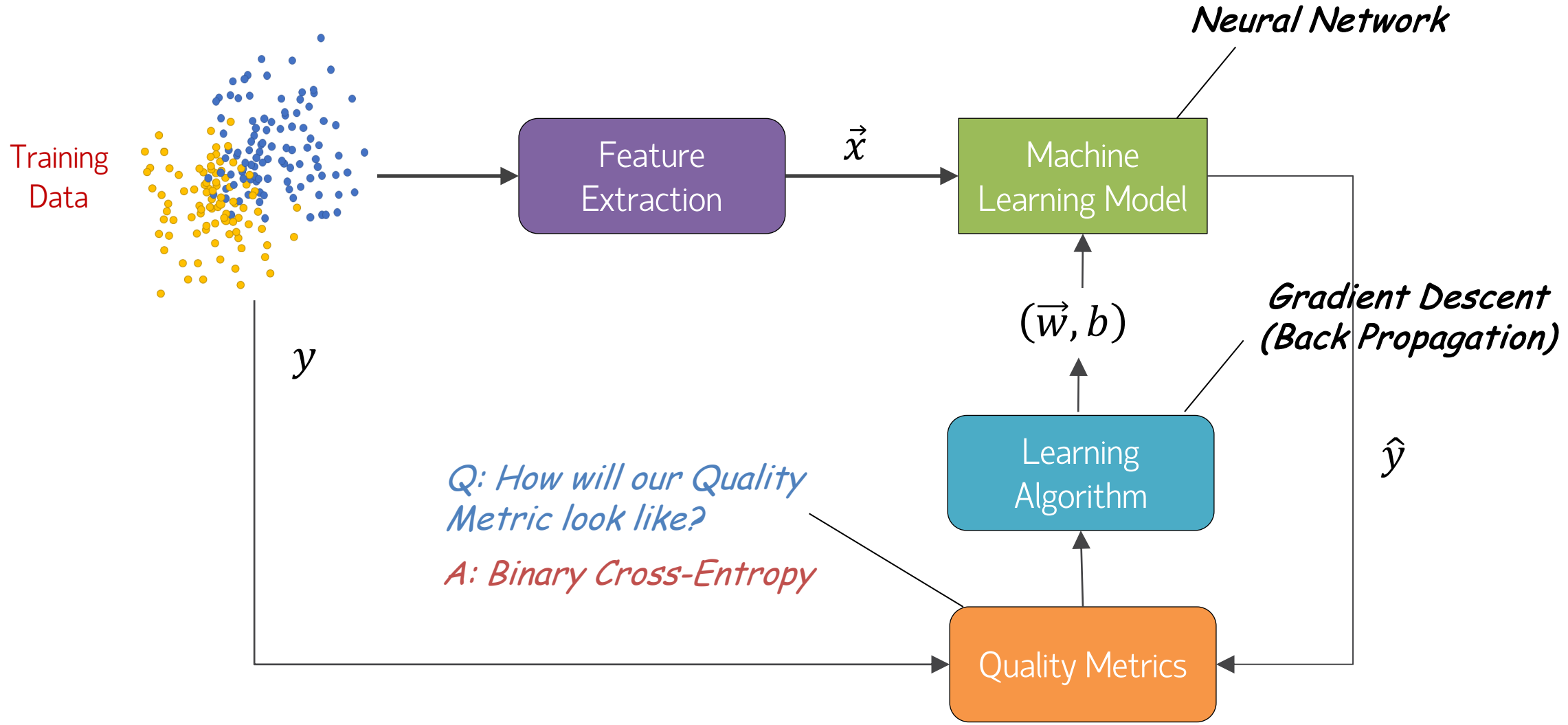


$\vec{w}^T \times \vec{x} + b = 0$

(Decision Boundary)

If $y^{(k)}$ is 1, then we'd like $P(1|\vec{x}^{(k)},\vec{w}) \rightarrow 1$.

Workflow: Binary Classification



Quality Metric: Likelihood Function

Likelihood Function:

# of Data Points in Training Dataset

If $y^{(i)}$ is 1, then we'd like $P(1|\vec{x}^{(i)}, \vec{w}, b) \rightarrow 1$.

$$L(\vec{w}, b) = \prod_{i=1}^{B} P(1|\vec{x}^{(i)}, \vec{w}, b)^{y^{(i)}} \times [1 - P(1|\vec{x}^{(i)}, \vec{w}, b)]^{(1-y^{(i)})}$$

$= 1$ if $y^{(i)} = 0$

$= 1$ if $y^{(i)} = 1$

If $y^{(i)}$ is 0, then we'd like $P(1|\vec{x}^{(i)}, \vec{w}, b) \rightarrow 0$ (or $P(0|\vec{x}^{(i)}, \vec{w}, b) \rightarrow 1$).

Note: $P(0|\vec{x}^{(i)}, \vec{w}, b) = 1 - P(1|\vec{x}^{(i)}, \vec{w}, b)$

Recall $p^0 = 1$ and $p^1 = p$.

On/Off Likelihood for 0/1 Labels

Likelihood Function:

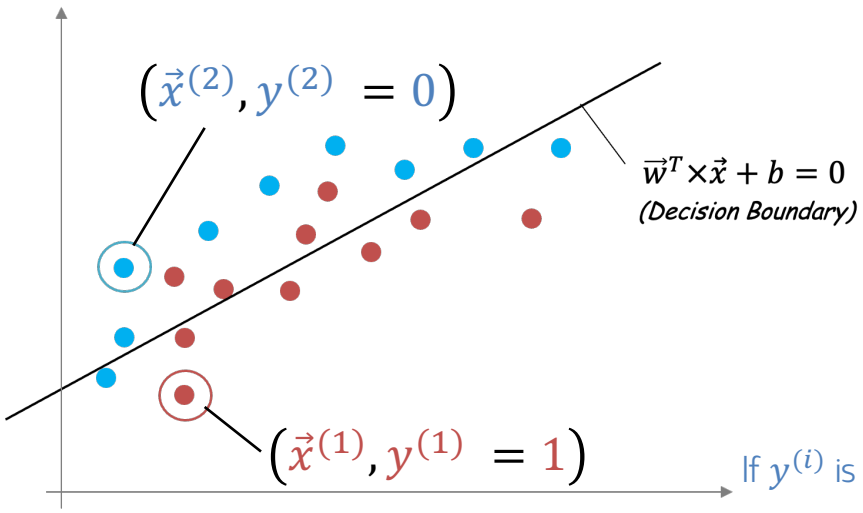
1 (OFF) if $y^{(i)} = 0$ 1 (OFF) if $y^{(i)} = 1$

$$L(\vec{w}, b) = \prod_{i=1}^{B} P(1|\vec{x}^{(i)}, \vec{w})^{y^{(i)}} \times [1 - P(1|\vec{x}^{(i)}, \vec{w})]^{(1-y^{(i)})}$$

If $y^{(i)}$ is 1, then we'd like $P(1|\vec{x}^{(i)}, \vec{w}) \rightarrow 1$.

$$\begin{aligned} L(\vec{x}^{(1)}, \vec{w}, b) &= P(1|\vec{x}^{(1)}, \vec{w}, b)^{y^{(1)}} \times [1 - P(1|\vec{x}^{(1)}, \vec{w}, b)]^{(1-y^{(1)})} \\ &= P(1|\vec{x}^{(1)}, \vec{w}, b)^{1} \times [1 - P(1|\vec{x}^{(1)}, \vec{w}, b)]^{(1-1)} \\ &= P(1|\vec{x}^{(1)}, \vec{w}, b) \end{aligned}$$

$$\begin{aligned} L(\vec{x}^{(2)}, \vec{w}, b) &= P(1|\vec{x}^{(2)}, \vec{w}, b)^{y^{(2)}} \times [1 - P(1|\vec{x}^{(2)}, \vec{w}, b)]^{(1-y^{(2)})} \\ &= P(1|\vec{x}^{(2)}, \vec{w}, b)^{0} \times [1 - P(1|\vec{x}^{(2)}, \vec{w}, b)]^{(1-0)} \\ &= 1 - P(1|\vec{x}^{(2)}, \vec{w}, b) \\ &= P(0|\vec{x}^{(2)}, \vec{w}, b) \end{aligned}$$



If $y^{(i)}$ is 0, then we'd like $P(1|\vec{x}^{(i)}, \vec{w}, b) \rightarrow 0$ (or $P(0|\vec{x}^{(i)}, \vec{w}, b) \rightarrow 1$).

Note: $P(0|\vec{x}^{(i)}, \vec{w}, b) = 1 - P(1|\vec{x}^{(i)}, \vec{w}, b)$

Related Properties:. $p^1 = p \mid p^0 = 1 \mid 1 \times p = p$

Log Likelihood Function

Likelihood Function:

$$L(\vec{w}, b) = \prod_{i=1}^{B} P(1|\vec{x}_i, \vec{w}, b)^{y_i} \times [1 - P(1|\vec{x}_i, \vec{w}, b)]^{(1-y_i)}$$

Log Likelihood Function:

$$\log L(\vec{w}, b) = \sum_{i=1}^{B} [y_i \log P(1|\vec{x}_i, \vec{w}, b) + (1 - y_i) \log(1 - P(1|\vec{x}_i, \vec{w}, b))]$$

Q: Why are we using log probabilities?

A: … Numerical Underflow …

Relevant Logarithmic Rules:

$$\log(A \times B) = \log A + \log B \qquad \log A^N = N \log A$$

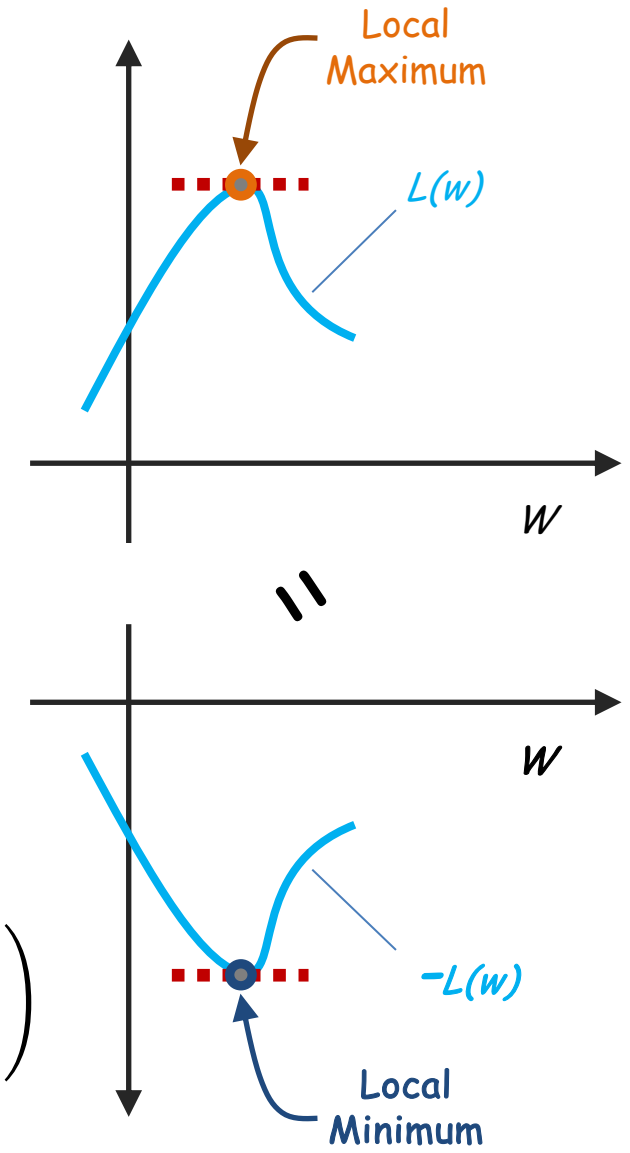
Gradient Descent: Minimisation

Maximising Log Likelihood:

$$\max_{\vec{w},b} \sum_{i=1}^{B} \left[y^{(i)} \log P(1|\vec{x}^{(i)}, \vec{w}, b) + (1 - y^{(i)}) \log(1 - P\left(1\middle|\vec{x}^{(i)}, \vec{w}, b\right)) \right]$$

Minimising Negative Log Likelihood:

$$\min_{\vec{w},b} -\left(\sum_{i=1}^{B} \left[y^{(i)} \log P(1|\vec{x}^{(i)}, \vec{w}, b) + (1 - y^{(i)}) \log(1 - P\left(1\middle|\vec{x}^{(i)}, \vec{w}, b\right)) \right] \right)$$



Binary Cross-Entropy

Symbol	Meaning
$y \in \{0,1\}$	Ground-Truth Label (1 = positive, 0 = negative)
$\hat{y} \in (0,1)$	Model's Predicted Probability for the Positive Class (Sigmoid of the Logit s)

Minimising Negative Log Likelihood:

$$\min_{\vec{w},b} -\left(\sum_{i=1}^{N}\left[y^{(i)} \log P(1|\vec{x}^{(i)},\vec{w},b) + (1-y^{(i)}) \log(1 - P\left(1\middle|\vec{x}^{(i)},\vec{w},b\right))\right]\right)$$

=

Binary Cross-Entropy in Neural Networks

Other Interpretations: Binary Cross-Entropy measures the difference from the model distribution $\hat{y}$ toward the true distribution y.

Binary Cross-Entropy as an On/Off Switch

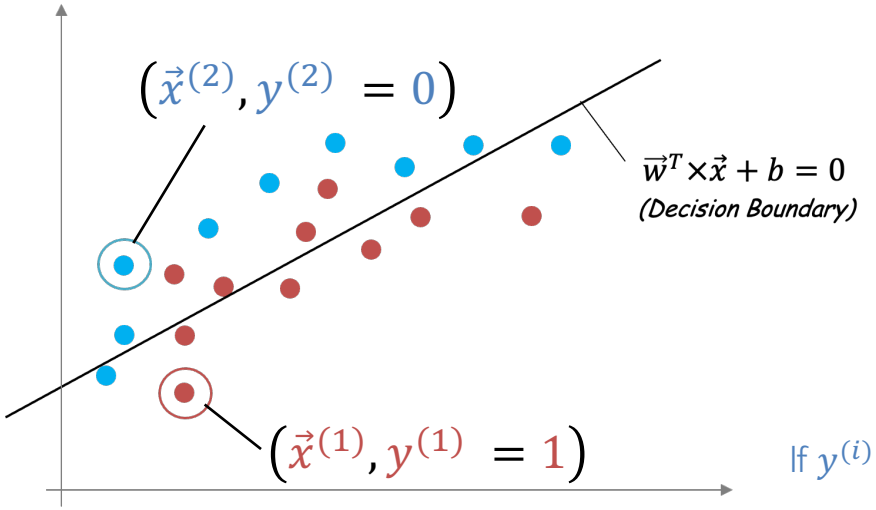
Binary Cross-Entropy:

0 (OFF) if $y^{(i)} = 0$ // 0 (OFF) if $y^{(i)} = 1$ //

$$L(\vec{w}, b) = -\sum_{i=1}^{N} \left[y^{(i)} \log P(1|\vec{x}^{(i)}, \vec{w}, b) + (1 - y^{(i)}) \log(1 - P(1|\vec{x}^{(i)}, \vec{w}, b)) \right]$$

Related Properties:

$P(y = 1\|\vec{x}, \vec{w}, b) \in (0,1)$	$P(y = 1\|\vec{x}, \vec{w}, b) \to 1^-$
$\log P(y = 1\|\vec{x}, \vec{w}, b) \in (-\infty, 0)$	$\log P(y = 1\|\vec{x}, \vec{w}, b) \to 0^-$
$-\log P(y = 1\|\vec{x}, \vec{w}, b) \in (0, +\infty)$	$-\log P(y = 1\|\vec{x}, \vec{w}, b) \to 0^+$



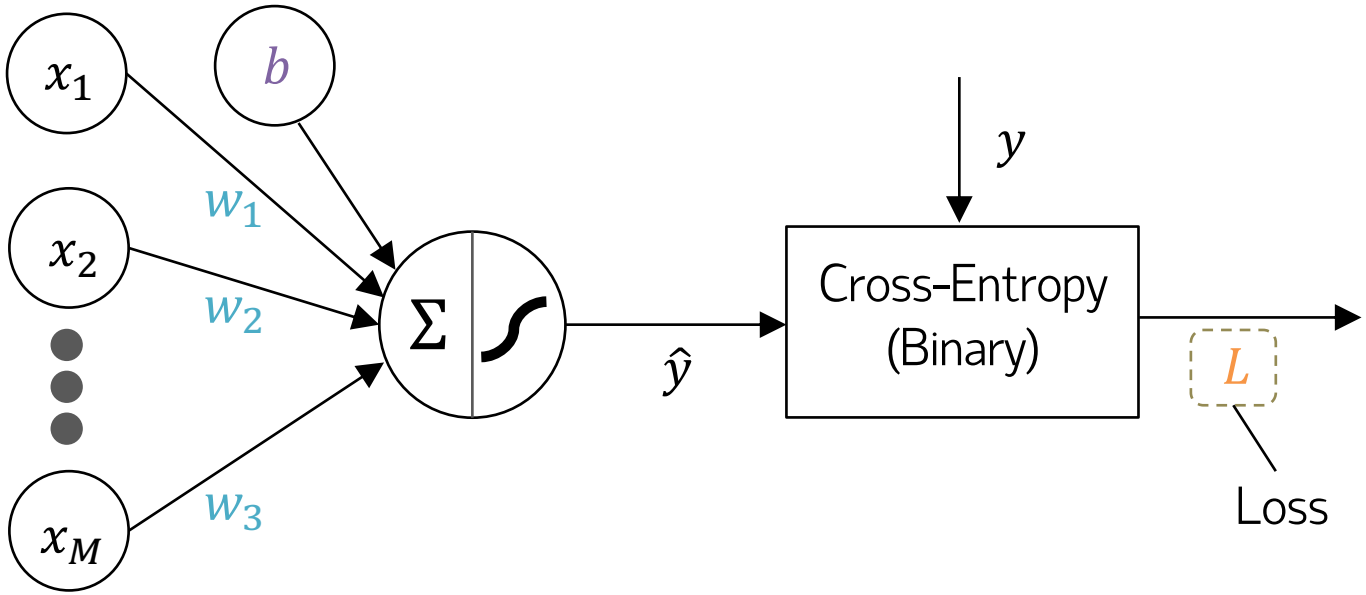
If $y^{(i)}$ is 1, then $-\log P(1|\vec{x}^{(i)}, \vec{w}) \to 0^+$.

$$\begin{aligned} L(\vec{x}^{(1)}, \vec{w}, b) &= -\left[y^{(1)} \log P(1|\vec{x}^{(1)}, \vec{w}, b) + (1 - y^{(1)}) \log\left(1 - P(1|\vec{x}^{(1)}, \vec{w}, b)\right) \right] \\ &= -\left[1 \times \log P(1|\vec{x}^{(1)}, \vec{w}, b) + (1 - 1) \log\left(1 - P(1|\vec{x}^{(1)}, \vec{w}, b)\right) \right] \\ &= -\log P(1|\vec{x}^{(1)}, \vec{w}, b) \end{aligned}$$

$$\begin{aligned} L(\vec{x}^{(2)}, \vec{w}, b) &= -\left[y^{(2)} \log P(1|\vec{x}^{(2)}, \vec{w}, b) + (1 - y^{(2)}) \log\left(1 - P(1|\vec{x}^{(2)}, \vec{w}, b)\right) \right] \\ &= -\left[0 \times \log P(1|\vec{x}^{(2)}, \vec{w}, b) + (1 - 0) \log\left(1 - P(1|\vec{x}^{(2)}, \vec{w}, b)\right) \right] \\ &= -\log\left(1 - P(1|\vec{x}^{(2)}, \vec{w}, b)\right) \\ &= -\log P(0|\vec{x}^{(2)}, \vec{w}, b) \end{aligned}$$

If $y^{(i)}$ is 0, then $-\log\left(1 - P(1|\vec{x}^{(2)}, \vec{w}, b)\right) \to 0^+$ (or $-\log P(0|\vec{x}^{(i)}, \vec{w}, b) \to 0^+$).

Need-to-Know: Parameters vs Loss



***Goal:** Bring the loss L down.*

***Q:** How will L be influenced by w_i and b?*

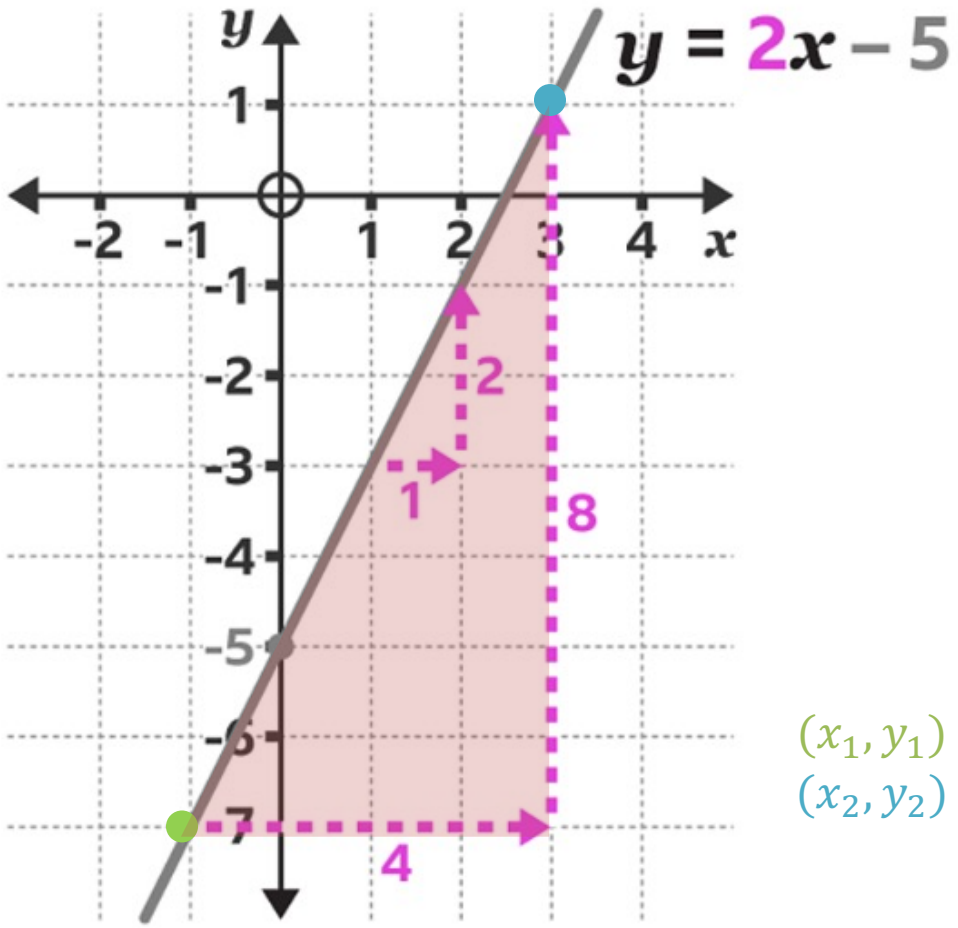
Forward Path:

$$\hat{y} = \sigma(\vec{w}^T \times \vec{x} + b\ )$$

Binary-Cross Entropy:

$$L(\vec{w}, b) = -\frac{1}{B}\sum_{k=1}^{B}\left[y^{(i)} \log \hat{y}^{(k)} + (1 - y^{(k)}) \log(1 - \hat{y}^{(k)})\right]$$

Linear Equation Graph



$(x_1, y_1) = (-1, -7)$

$(x_2, y_2) = (3, 1)$

Intercept

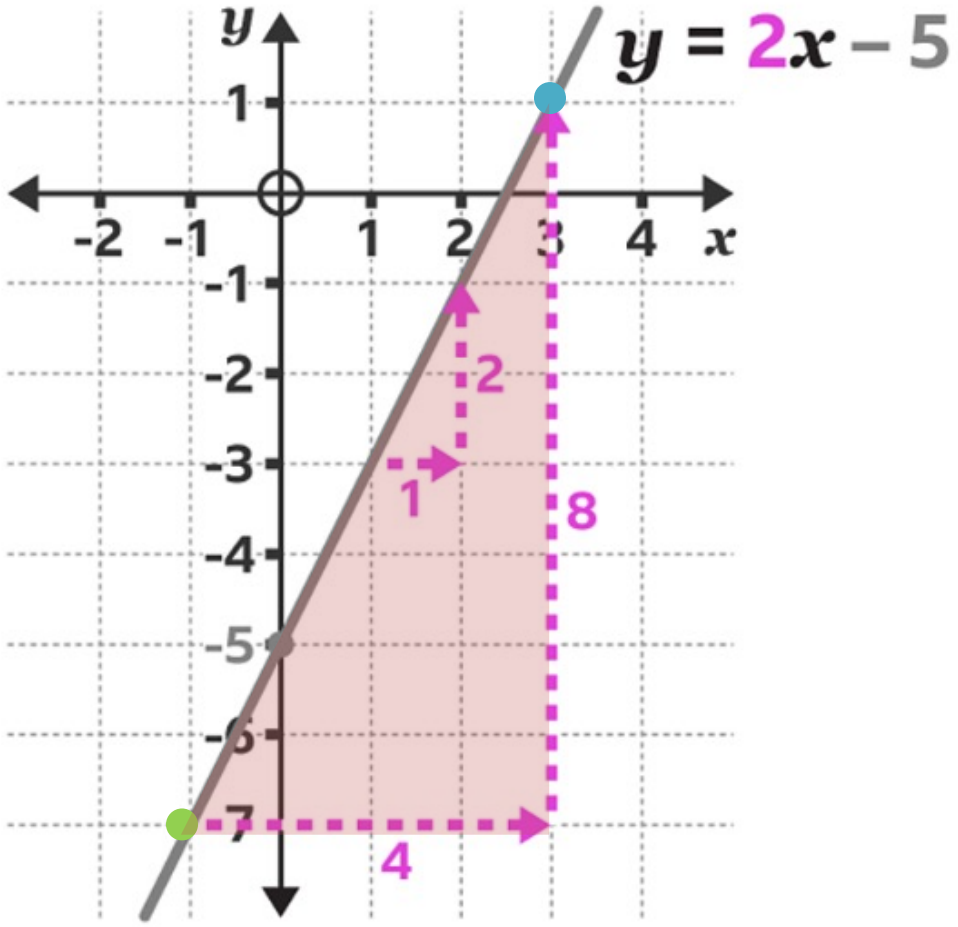
$y = mx + c$

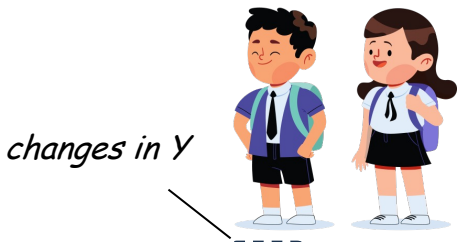
Gradient (or Slope)

$$\text{Gradient} = \frac{y_2 - y_1}{x_2 - x_1} = \frac{1 - (-7)}{3 - (-1)} = 2$$

Intercept $= -5$

Gradient





Q1: What is the value of ΔY when ΔX = +1?

A1: +2 (= 2x(+1))

Q2: What is the value of ΔY when ΔX = +4?

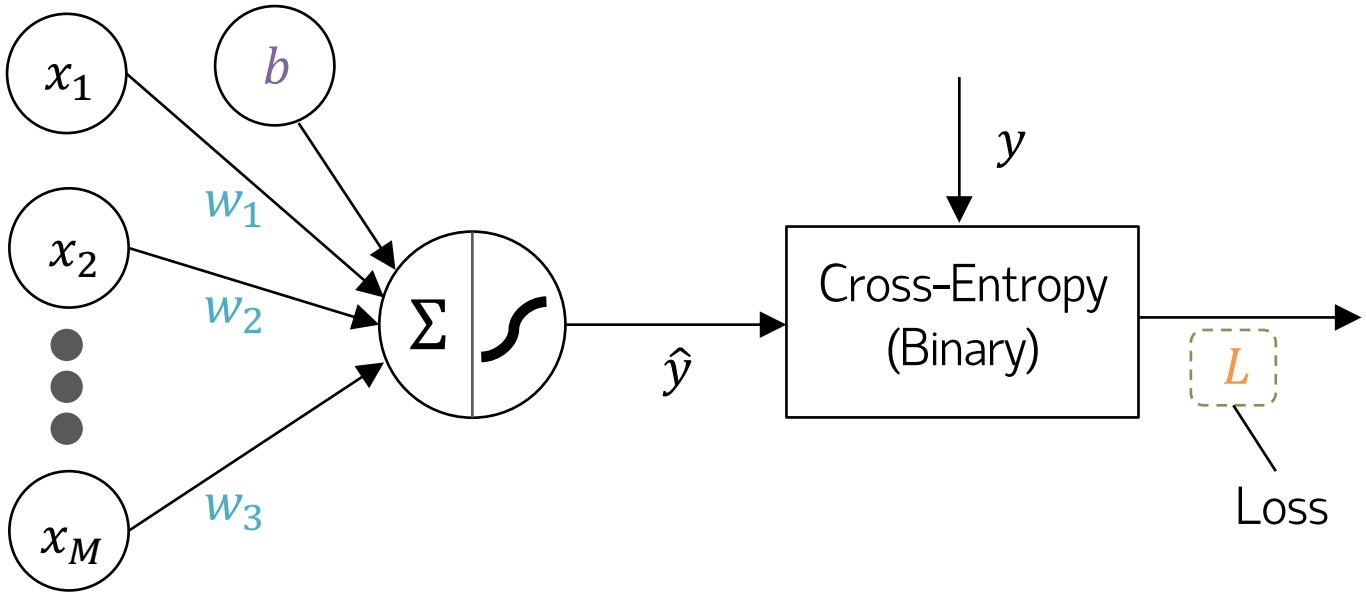
A2: +8 (= 2x(+4))

Q3: What is the value of ΔY when ΔX = -2?

A3: -4 (= 2x(-2))

2 is in fact our gradient $\frac{\partial y}{\partial x}$. *Intuitively,* $\frac{\partial y}{\partial x}$ *defines how ΔY is influenced by ΔX.*

Need-to-Know: Parameters vs Loss (cont.)



Goal: Bring the loss L down.

Q: How will L be influenced by w_i and b?

A: Gradients, i.e. $\frac{\partial L}{\partial w_i}$ and $\frac{\partial L}{\partial b}$.

Forward Path:

$$\hat{y} = \sigma(\vec{w}^T \times \vec{x} + b\ )$$

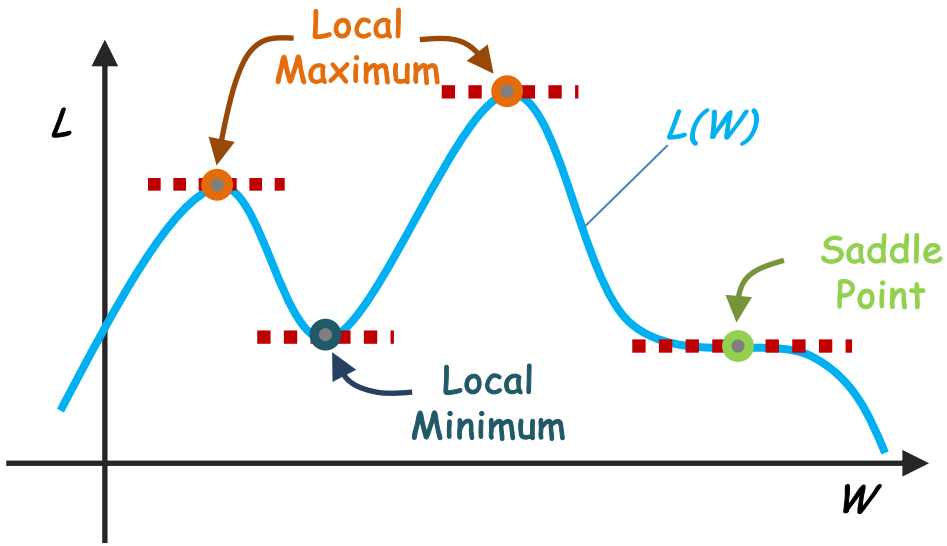
Binary-Cross Entropy:

$$L(\vec{w}, b) = -\frac{1}{B}\sum_{k=1}^{B}\left[y^{(i)} \log \hat{y}^{(k)} + (1 - y^{(k)}) \log(1 - \hat{y}^{(k)})\right]$$

Best Decision Boundary

Miminising Binary Cross-Entropy:

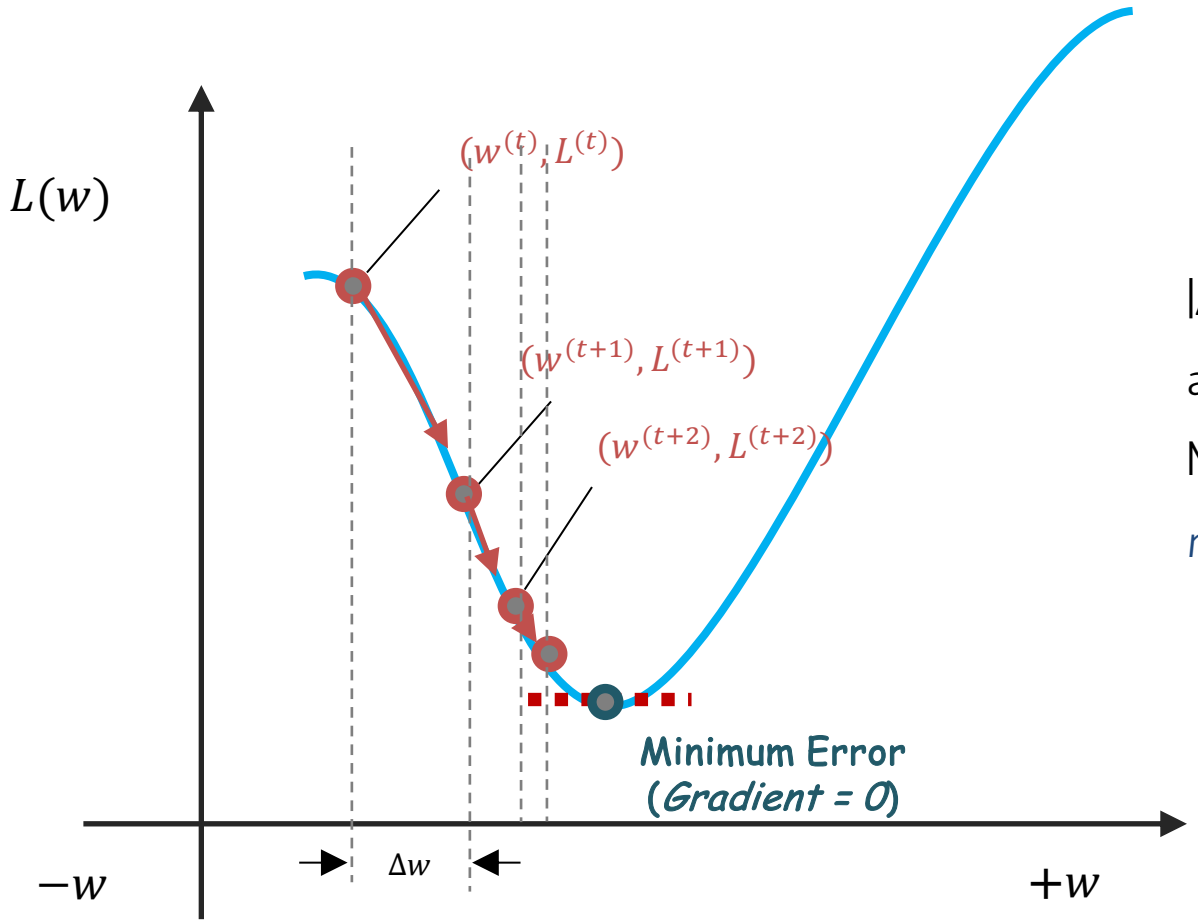
$$\max_{\vec{w},b} -\sum_{i=1}^{B}\left[y^{(i)}\log P(1|\vec{x}^{(i)},\vec{w},b) + (1-y^{(i)})\log(1-P\left(1\middle|\vec{x}^{(i)},\vec{w},b\right))\right]$$



Q: Which value of w will L(w) be minimum?

A: … Slope (or Gradient) = 0 …

Recall: Gradient Descent



Learning Rate

$$w^{(t+1)} \leftarrow w^{(t)} - \eta \times \frac{\partial L(w^{(t)})}{\partial w}$$

$|\Delta w|$ is *large* when w is further away from the *minimum* point as $\left|\frac{\partial L(w^{(t)})}{\partial w}\right|$ large.

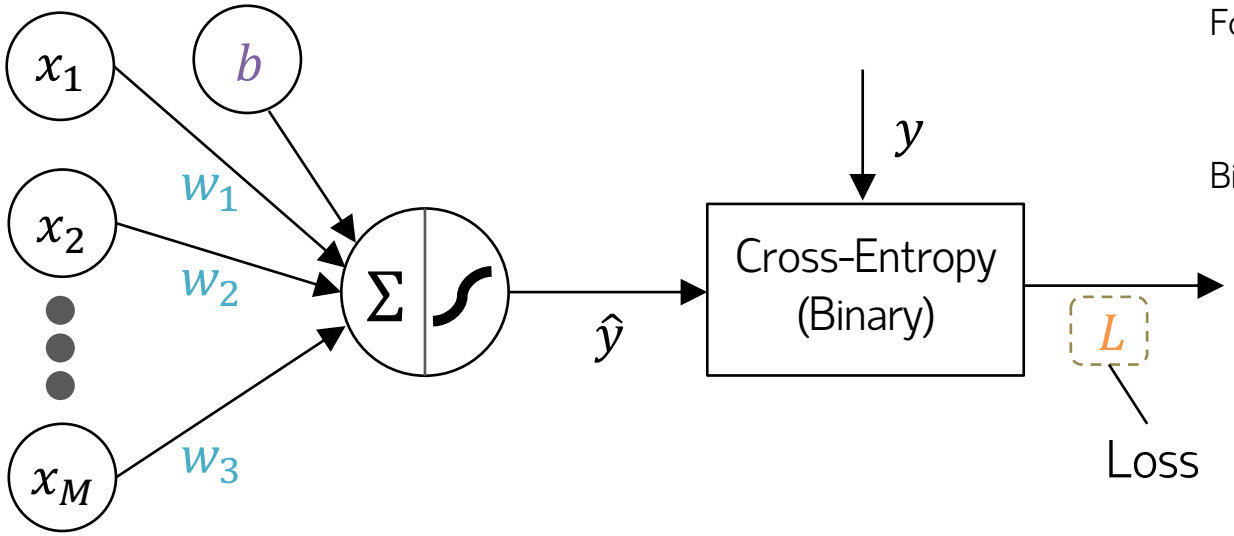
Meanwhile, $|\Delta w|$ becomes smaller when w is closer to the *minimum* point as $\left|\frac{\partial L(w^{(t)})}{\partial w}\right|$ is small.

Step:

$$|\Delta w| = \left|\eta \times \frac{\partial L(w^{(t)})}{\partial w}\right|$$

$\text{Sign}(\frac{\partial L(w^{(t)})}{\partial w})$ → Direction | $|\frac{\partial L(w^{(t)})}{\partial w}|$ → *Step Size* | η → *Convergent Time*

Gradients



Forward Path:

$$\hat{y} = \sigma(\vec{w}^T \times \vec{x} + b)$$

Binary-Cross Entropy:

$$L(\vec{w}, b) = -\sum_{k=1}^{B} \left[y^{(i)} \log \hat{y}^{(k)} + (1 - y^{(k)}) \log(1 - \hat{y}^{(k)}) \right]$$

Goal: *Bring the loss* L *down.*

Changing w_i *and* b *affect the output* $\hat{y}$*; that in turn will change the loss* L*.*

Q: How to see the changes?

A: Gradients, of course. Chain-Rule: "Loss → Output → Weight/Bias"

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w_i} \quad \text{and} \quad \frac{\partial L}{\partial b} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial b}.$$

L is indirectly affected by w_i *and* b*.*

"How is w_i *affecting* $\hat{y}$*?"*

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w_i}$$

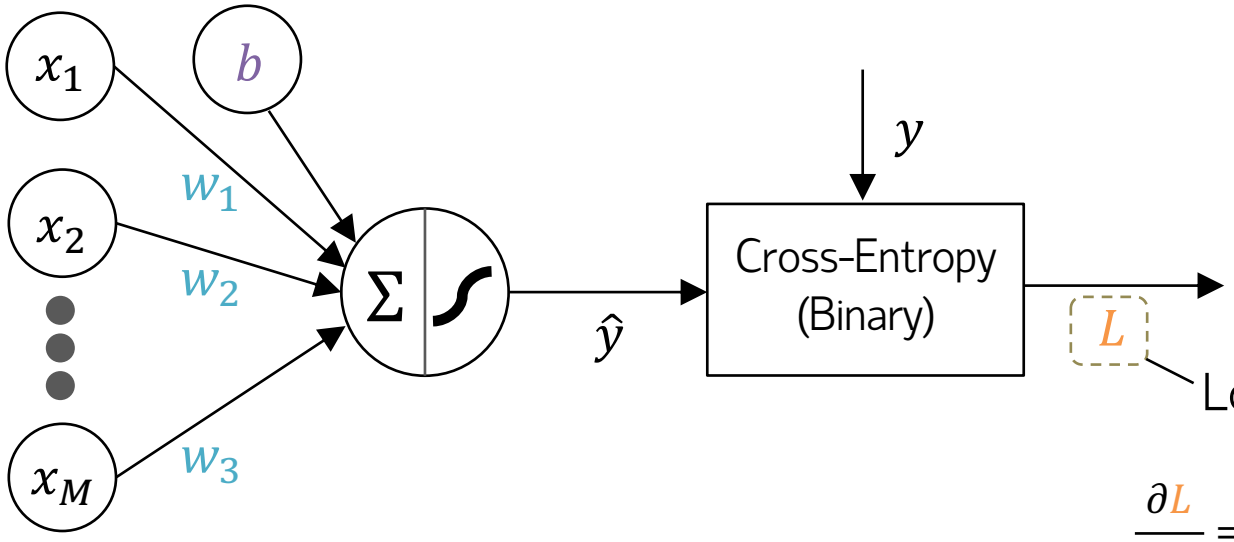
"How is $\hat{y}$ *affecting* L*?"*

"How is b *affecting* $\hat{y}$*?"*

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial b}$$

"How is $\hat{y}$ *affecting* L*?"*

Gradient Descent



Forward Path:

$$\hat{y} = \sigma(\vec{w}^T \times \vec{x} + b)$$

Binary-Cross Entropy:

$$L(\vec{w}, b) = -\sum_{k=1}^{B} \left[y^{(i)} \log \hat{y}^{(k)} + (1 - y^{(k)}) \log(1 - \hat{y}^{(k)}) \right]$$

"We compute $\frac{\partial L}{\partial \hat{y}}$ once, then reuse the value to obtain $\frac{\partial L}{\partial w_i}$ and $\frac{\partial L}{\partial b}$ for all weights and biases."

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w_i}$$

$$= \frac{(\hat{y} - y)}{\hat{y}(1 - \hat{y})} \hat{y}(1 - \hat{y}) x_i$$

$$= (\hat{y} - y) x_i$$

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial b}$$

$$= \frac{(\hat{y} - y)}{\hat{y}(1 - \hat{y})} \hat{y}(1 - \hat{y})$$

$$= \hat{y} - y$$

Update: Step against the Slope

$$w_i^{(t+1)} \leftarrow w_i^{(t)} - \eta \frac{\partial L}{\partial w_i}(w_i^{(t)})$$

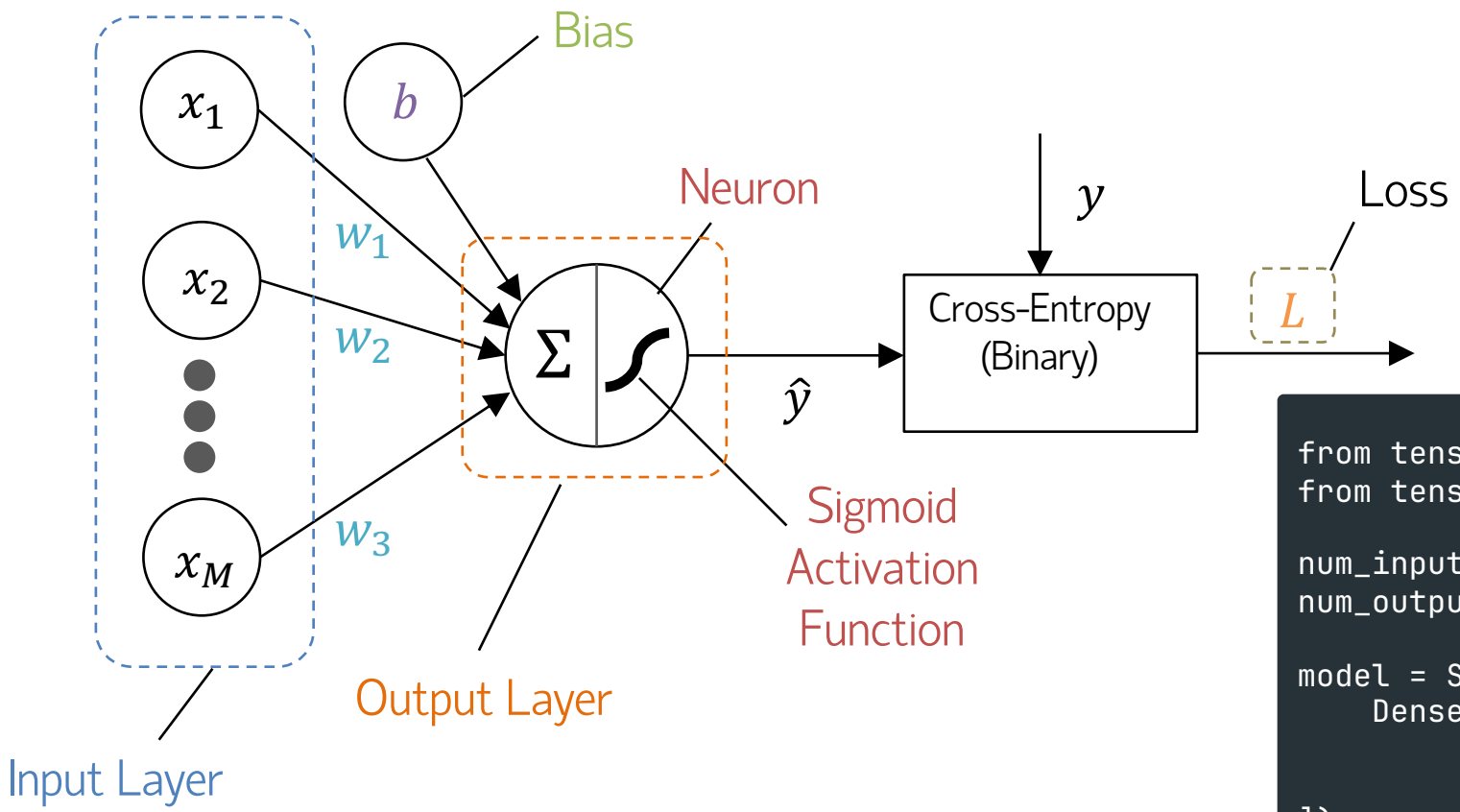
$$\leftarrow w_i^{(t)} - \eta \times \left(\sum_{k=1}^{B} (\hat{y}^{(k)} - y^{(k)}) x_i^{(k)} \right)$$

$$b^{(t+1)} \leftarrow b^{(t)} - \eta \frac{\partial L}{\partial b}(b^{(t)})$$

$$\leftarrow b^{(t)} - \eta \times \left(\sum_{k=1}^{B} (\hat{y}^{(k)} - y^{(k)}) \right)$$

Function	Derivative
$\sigma(z)$	$\sigma(z)(1 - \sigma(z))$
$\log \sigma(z)$	$(1 - \sigma(z))$

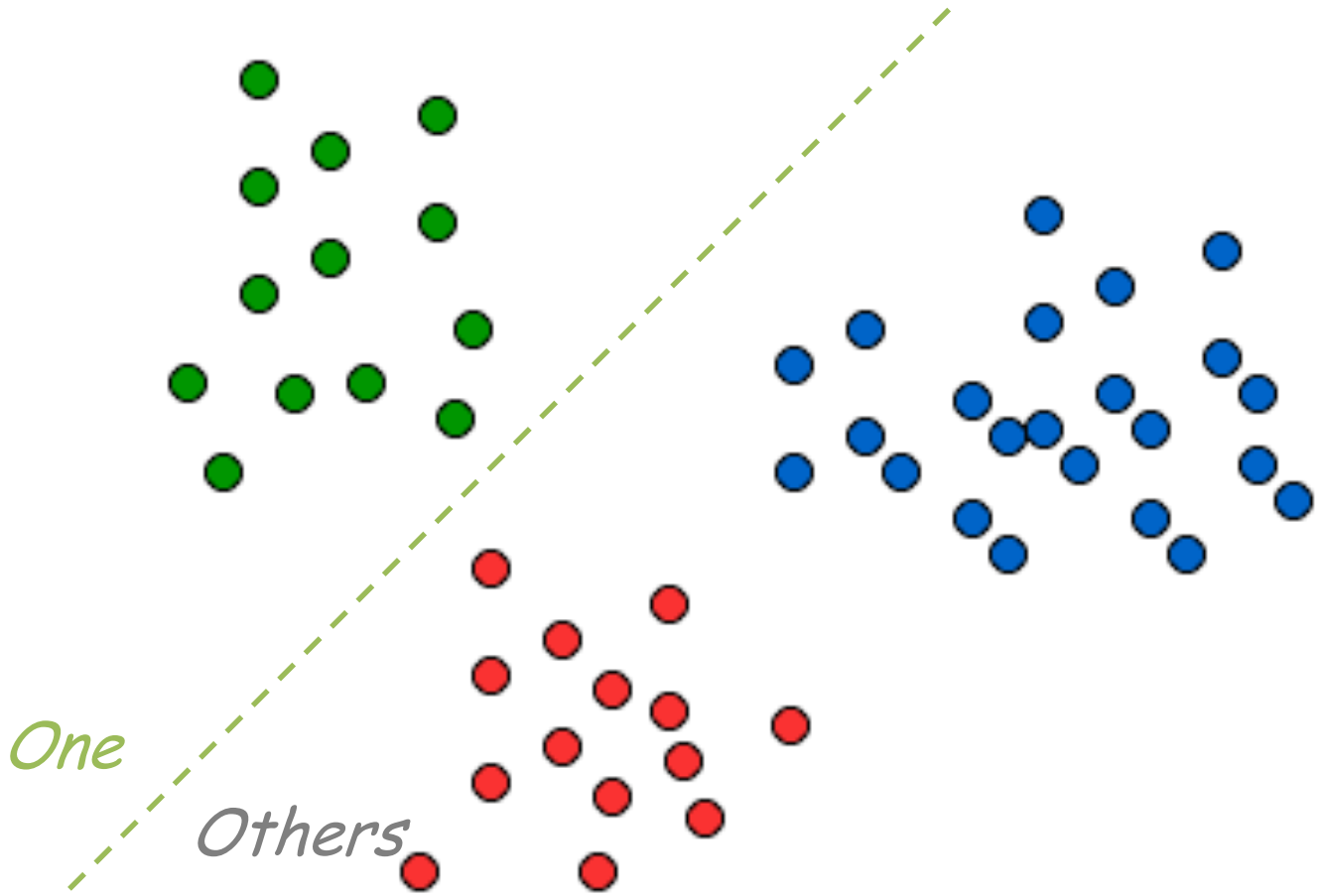
Tensorflow Code Snippet



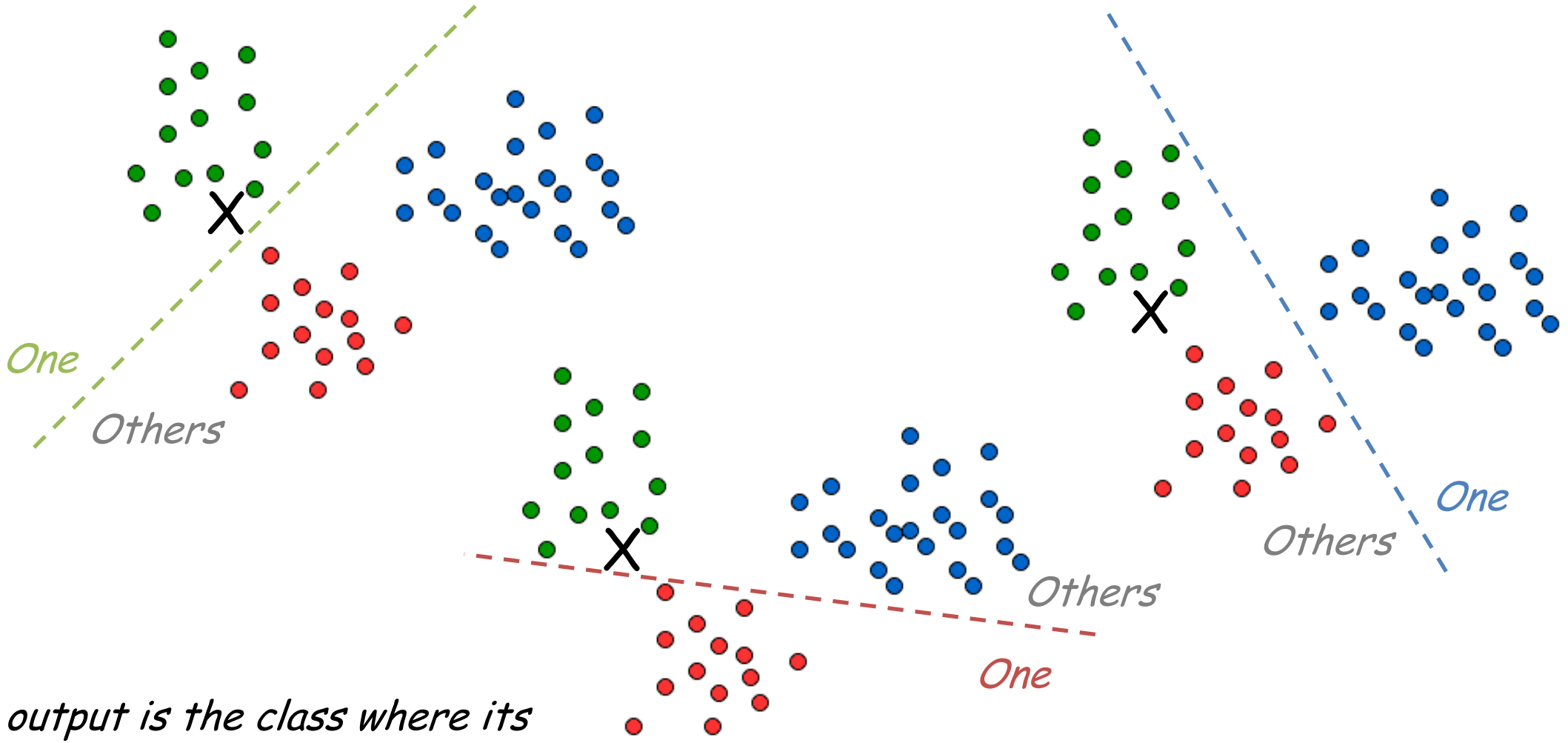

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense

num_inputs = M          # input dimension
num_outputs = 1         # input dimension

model = Sequential([
    Dense(num_outputs,
          activation="sigmoid”,
          input_shape=(num_inputs,))
])

model.compile(optimizer=”sgd",
              loss="binary_crossentropy",
              metrics=["accuracy"])
```

TensorFlow

Multiclass Classification



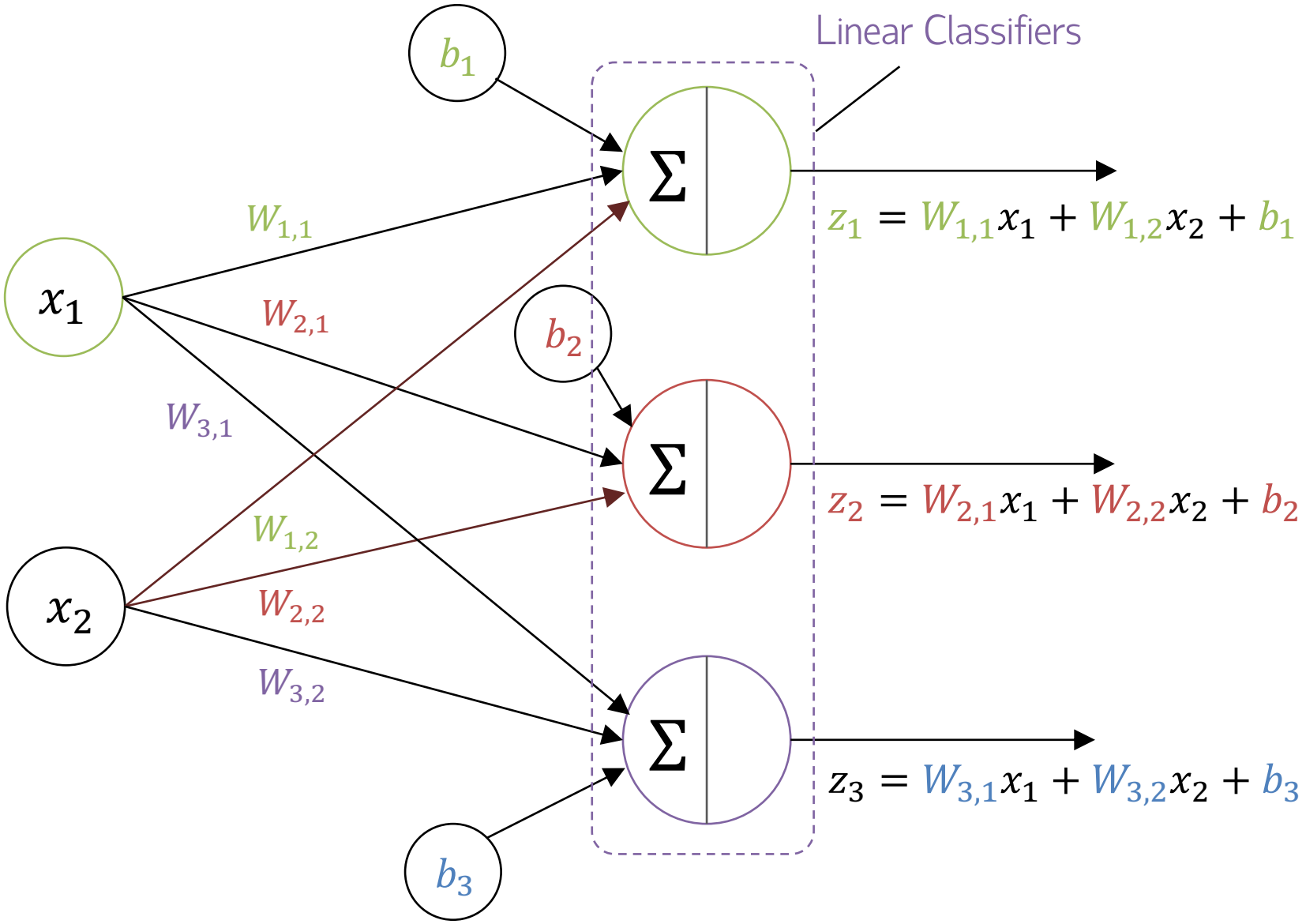
3 Linear Classifiers

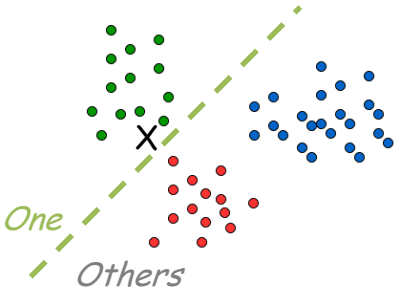


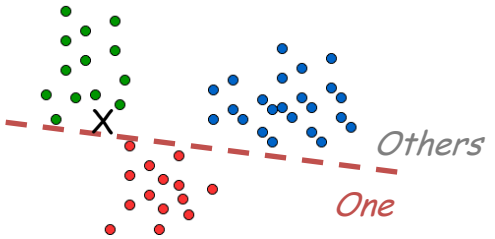
The output is the class where its normalised output probability is the highest (among the 3 classifiers).

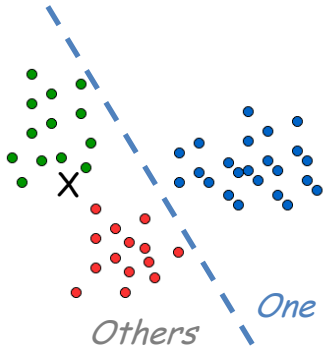
Ex. [0.6, 0.35, 0.05] → Green Class

Multi-Class Classifier: One vs Others

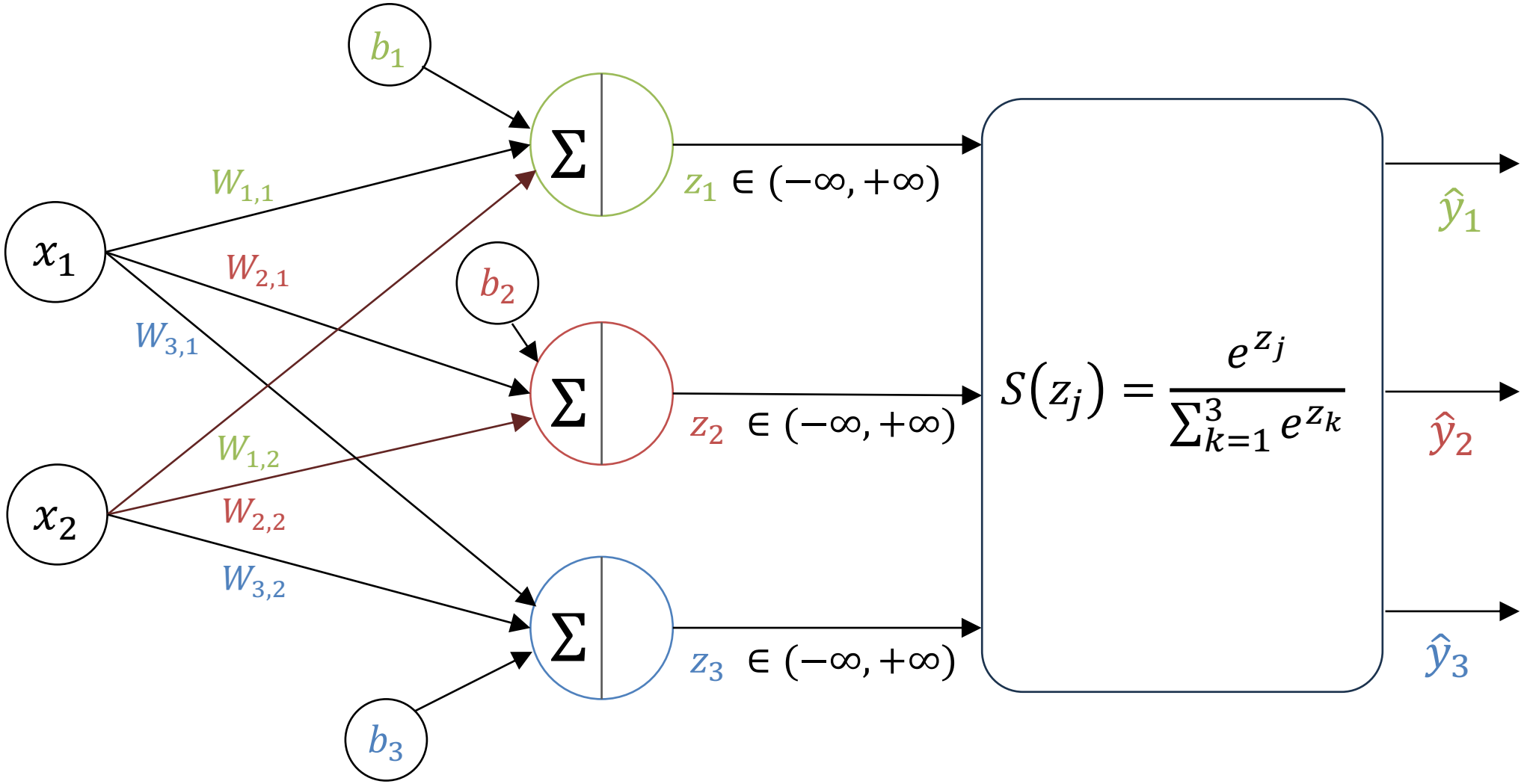






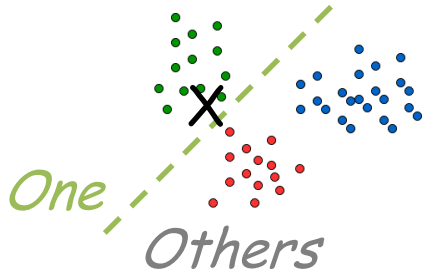


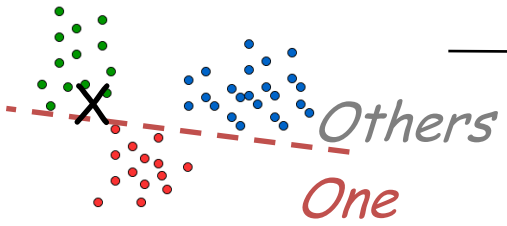
Softmax Function: Normalised Probabilities

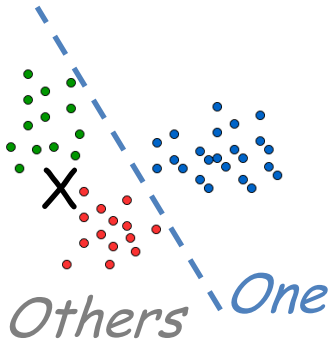


Normalised Probabiliies: $\hat{y}_1 + \hat{y}_2 + \hat{y}_3 = 1.$

SoftMax Function: Calculation Example







$z_1 = 1$

$z_2 = -1$

$z_3 = -2$

$$S(z_j) = \frac{e^{z_j}}{\sum_{k=1}^{3} e^{z_k}}$$

$$\hat{y}_1 = P(y_1 = 1|\vec{x}) = \frac{e^{1}}{e^{1} + e^{-1} + e^{-2}} = 0.844$$

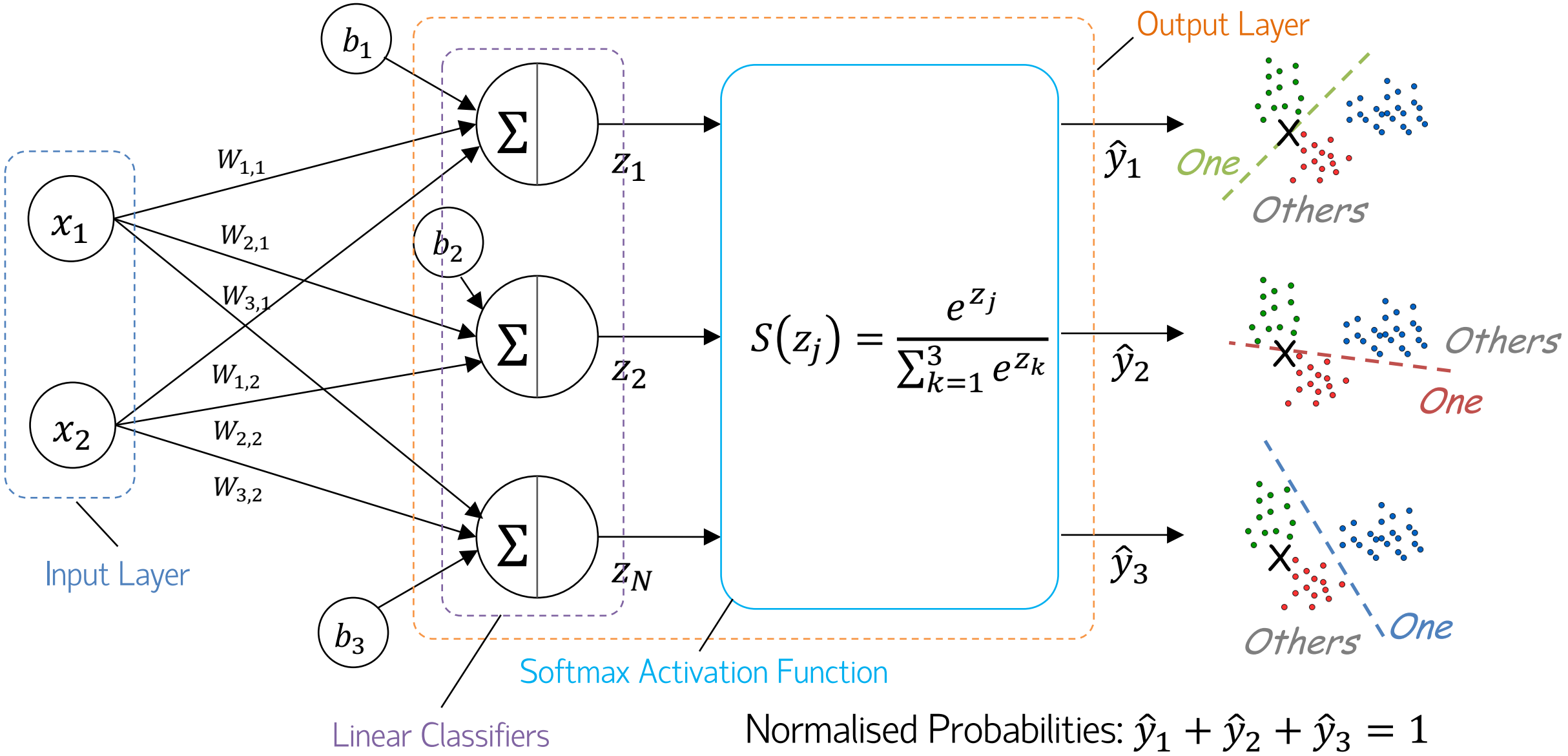
$$\hat{y}_2 = P(y_2 = 1|\vec{x}) = \frac{e^{-1}}{e^{1} + e^{-1} + e^{-2}} = 0.114$$

$$\hat{y}_3 = P(y_3 = 1|\vec{x}) = \frac{e^{-2}}{e^{1} + e^{-1} + e^{-2}} = 0.042$$

Normalisation: $\sum_{i=1}^{3} \hat{y}_i = 0.844 + 0.114 + 0.042 = 1.0$

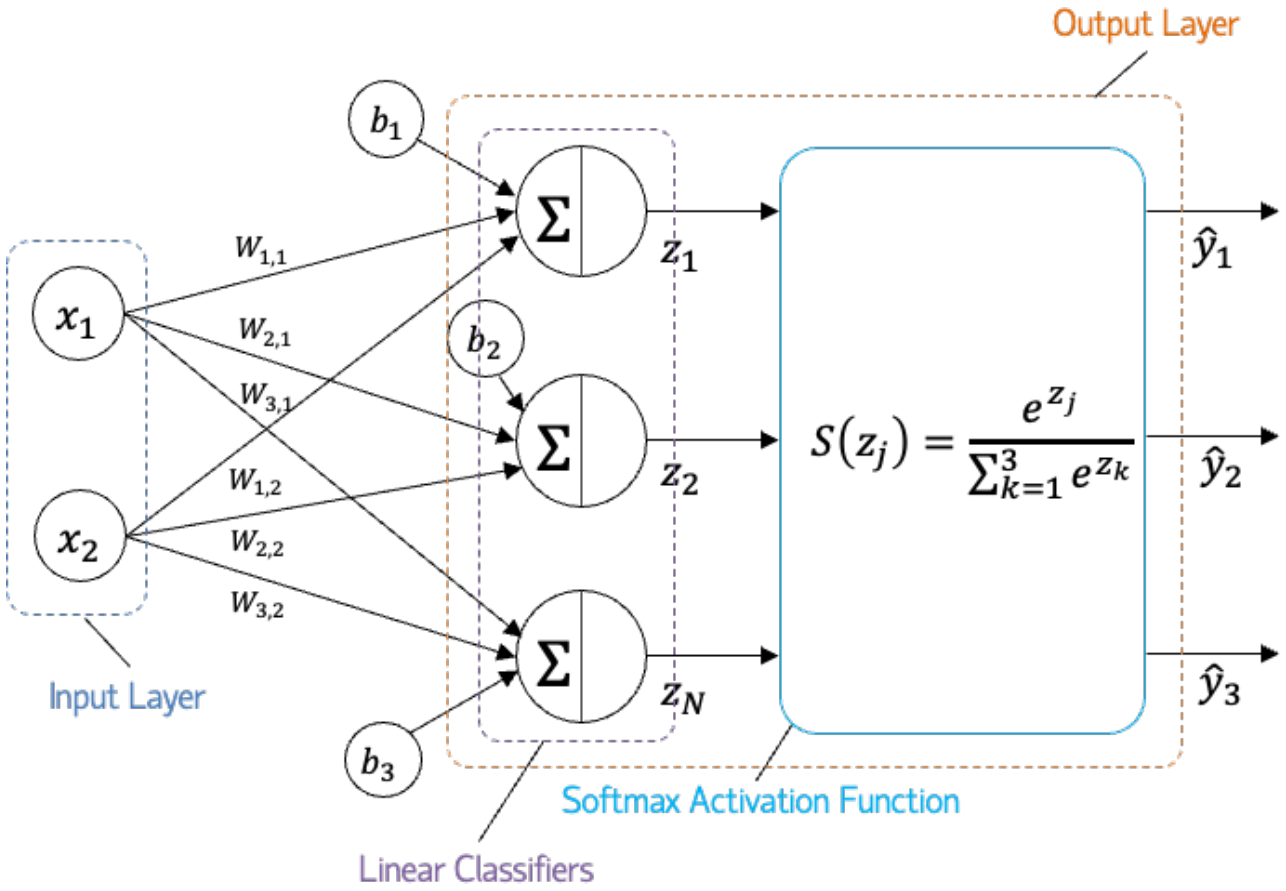
We predict the class where its normalised probability is the highest. Ex. [0.844, 0.114, 0.042] → Green Class

Multi-Class Classifier



Normalised Probabilities: $\hat{y}_1 + \hat{y}_2 + \hat{y}_3 = 1$

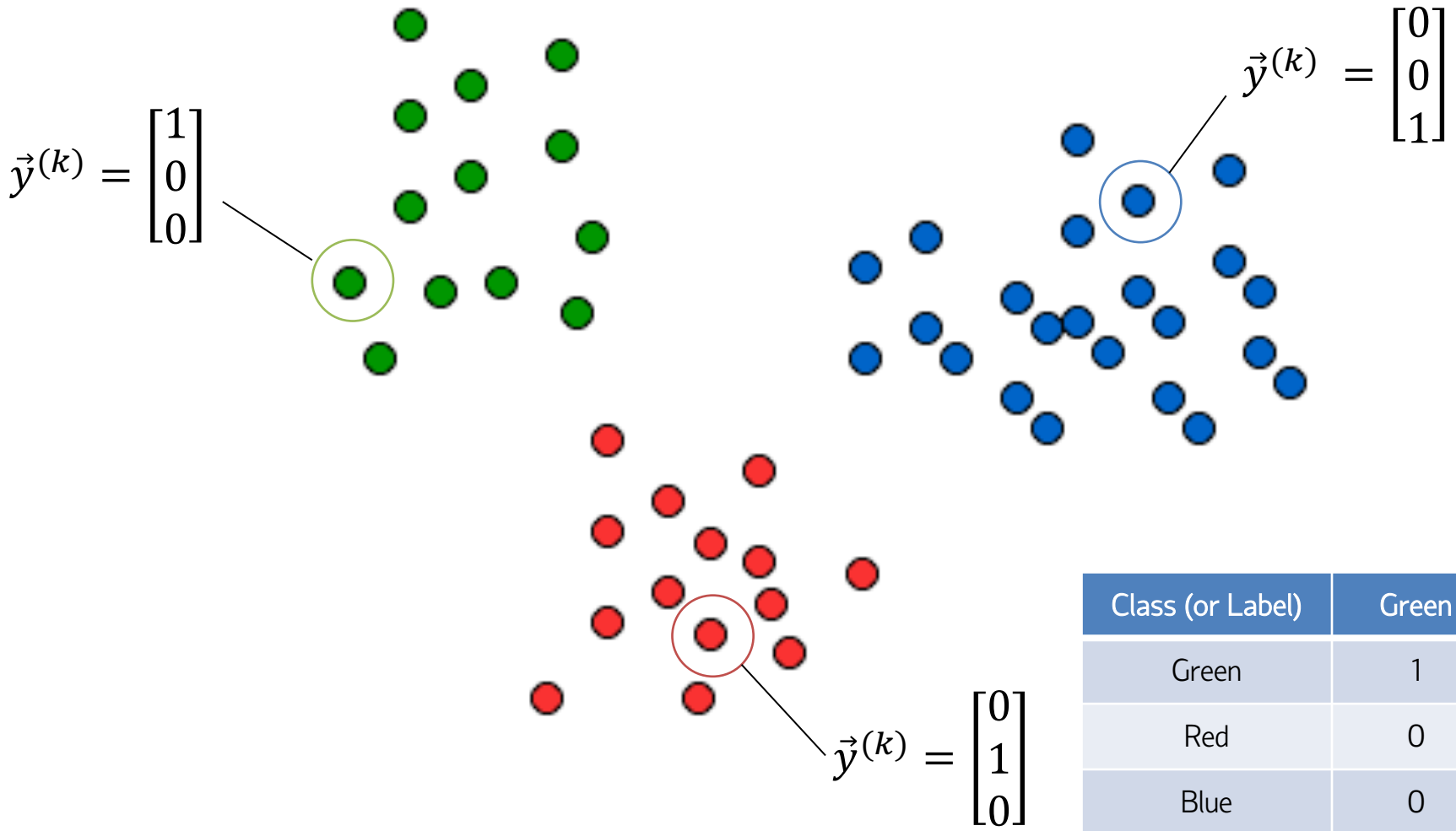
Tensorflow Code Snippet




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

num_inputs = 2          # input dimension
num_outputs = 3         # output dimension


model = Sequential([
    Dense(num_outputs,
          activation="softmax",
          input_shape=(num_inputs,))
])
```

TensorFlow

One-Hot Encoder



Class (or Label)	Green	Red	Blue
Green	1	0	0
Red	0	1	0
Blue	0	0	1

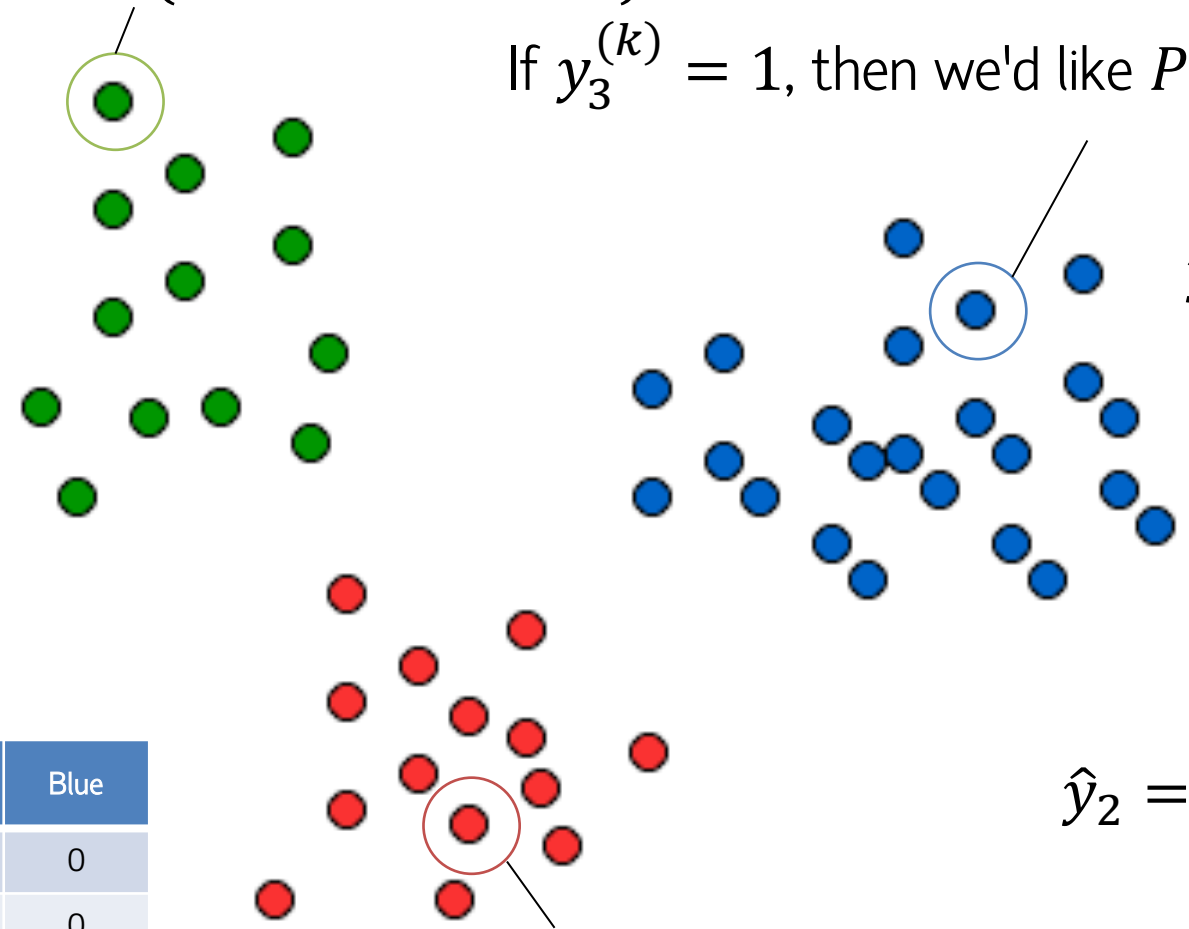
What We Are Optimising

$$\hat{y}_1 = S\left(W_{1,:} \times \vec{x}^{(k)} + b_1\right)$$

If $y_1^{(k)} = 1$, then we'd like $P\left(y_1 = 1 | \vec{x}^{(k)}, \vec{w}, b\right) \to 1$.

If $y_3^{(k)} = 1$, then we'd like $P\left(y_3 = 1 | \vec{x}^{(k)}, \vec{w}, b\right) \to 1$.

$$\hat{y}_3 = S\left(W_{3,:} \times \vec{x}^{(k)} + b_3\right)$$

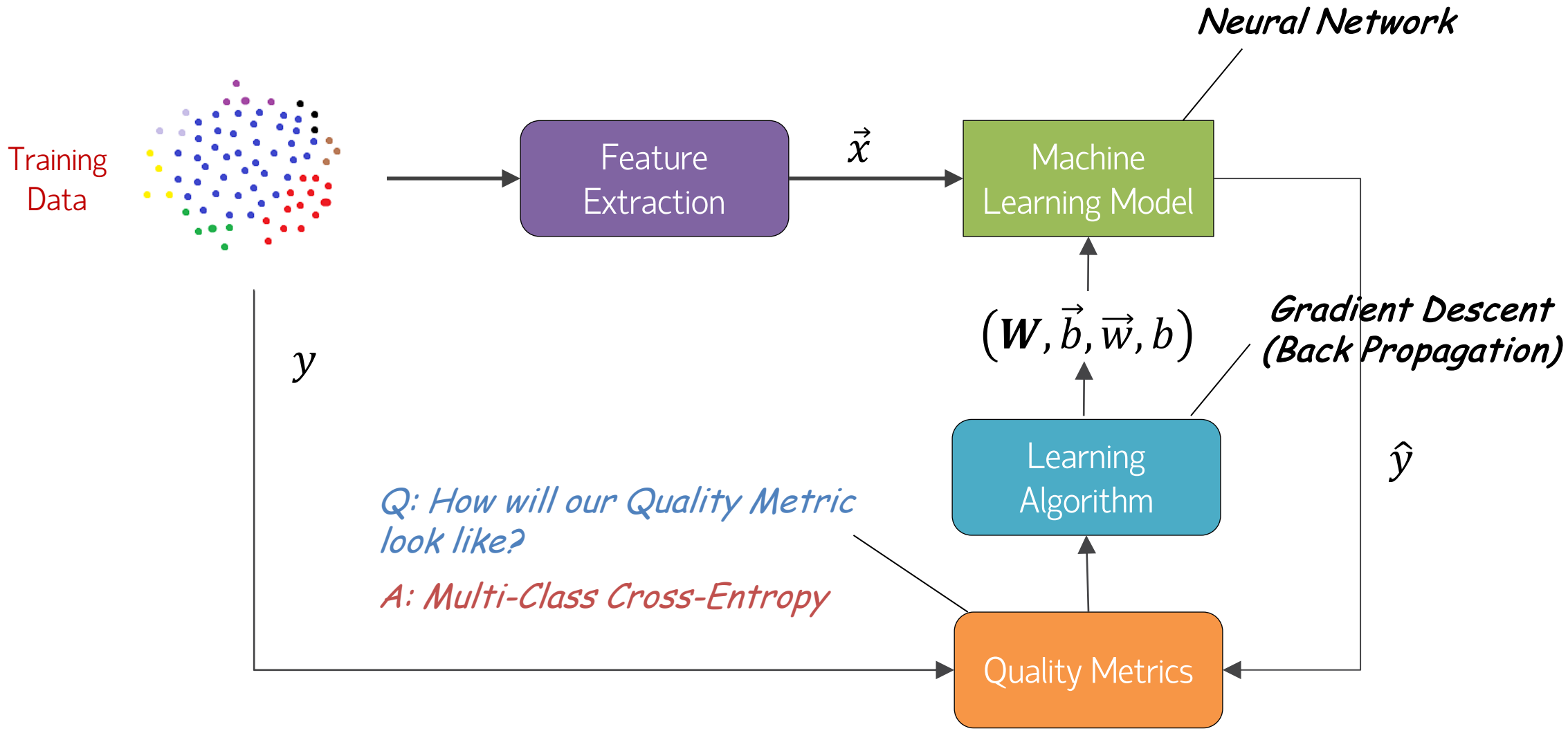


Class (or Label)	Green	Red	Blue
Green	1	0	0
Red	0	1	0
Blue	0	0	1

$$\hat{y}_2 = S\left(W_{2,:} \times \vec{x}^{(k)} + b_2\right)$$

If $y_2^{(k)} = 1$, then we'd like $P\left(y_2 = 1 | \vec{x}^{(k)}, \vec{w}, b\right) \to 1$.

Workflow: Multi-Class Classification



Quality Metric: Multi-Class Cross-Entropy

Multi-Class (Categorical) Cross-Entropy measures the discrepancy between the model's predicted probability vector $\hat{y}$ and the true one-hot class distribution $\vec{y}$ across all M classes.

Per-Sample Multi-Class Cross-Entropy (MCCE):

$$L_{MCCE}(\vec{y}, \hat{\boldsymbol{Y}}) = -\sum_{i=1}^{N} y_i \log \hat{y}_i$$

# of Classes → N

Symbol	Meaning
$i \in \{1, \dots, N\}$	Ground-truth class index (an integer that picks the correct class).
$y \in \{0,1\}^N$	Equivalent One-Hot vector form of the label where exactly one entry is 1 and the rest are 0.
$\hat{\boldsymbol{y}} = [\hat{y}_1, \dots, \hat{y}_N]$	Model's predicted class-probability vector produced by a soft-max layer; $\hat{y}_i$ is the predicted probability for class i, where $\sum_{i=1}^{N} \hat{y}_i = 1$
$\vec{z} = [z_1, \dots, z_N]$	Linear (un-normalised) scores before SoftMax: $\hat{y} = S(\vec{z})$.

Multi-Class Cross Entropy :

$$L(W, \vec{b}) = \frac{1}{B} \sum_{k=1}^{B} L_{MCCE}(\vec{y}^{(k)}, \hat{\boldsymbol{Y}}^{(k)})$$

# of Data Points in Training Dataset → B

Cross-Entropy as an On/Off Switch

Per Sample Cross-Entropy:

$$L_{MCCE}(\vec{y}, \hat{\boldsymbol{Y}}) = -\sum_{i=1}^{N} y_i \log \hat{y}_i$$

If $y_1^{(i)}$ is 1, then we'd like $-\log P(y_1 = 1|\vec{x}^{(i)}, \vec{w}, b) \to 0^+$.

$$L(\vec{x}^{(i)}, \vec{w}, b) = -\left[y_1^{(i)} \log \hat{y}_1^{(i)} + y_2^{(i)} \log \hat{y}_2^{(i)} + y_3^{(i)} \log \hat{y}_3^{(i)}\right]$$
$$= -\left[1 \times \log \hat{y}_1^{(i)} + 0 \times \log \hat{y}_2^{(i)} + 0 \times \log \hat{y}_3^{(i)}\right]$$
$$= -\log \hat{y}_1^{(i)}$$

If $y_2^{(j)}$ is 1, then we'd like $-\log P(y_1 = 1|\vec{x}^{(j)}, \vec{w}, b) \to 0^+$.

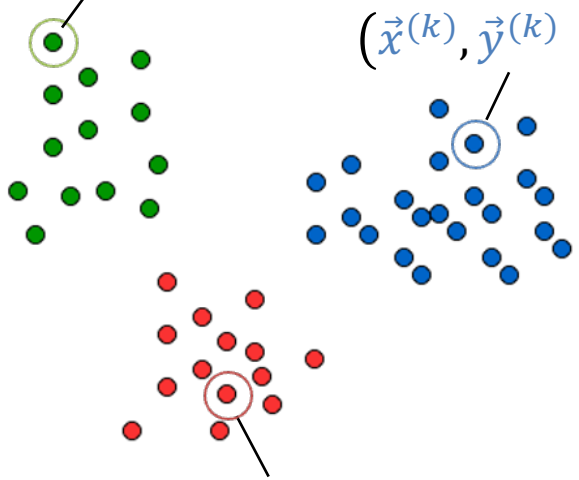
$$L(\vec{x}^{(j)}, \vec{w}, b) = -\left[y_1^{(j)} \log \hat{y}_1^{(j)} + y_2^{(j)} \log \hat{y}_2^{(j)} + y_3^{(j)} \log \hat{y}_3^{(j)}\right]$$
$$= -\left[0 \times \log \hat{y}_1^{(j)} + 1 \times \log \hat{y}_2^{(j)} + 0 \times \log \hat{y}_3^{(j)}\right]$$
$$= -\log \hat{y}_2^{(j)}$$

If $y_3^{(k)}$ is 1, then we'd like $-\log P(y_3 = 1|\vec{x}^{(k)}, \vec{w}, b) \to 0^+$.

$$L(\vec{x}^{(k)}, \vec{w}, b) = -\left[y_1^{(k)} \log \hat{y}_1^{(k)} + y_2^{(k)} \log \hat{y}_2^{(k)} + y_3^{(k)} \log \hat{y}_3^{(k)}\right]$$
$$= -\left[0 \times \log \hat{y}_1^{(k)} + 0 \times \log \hat{y}_2^{(k)} + 1 \times \log \hat{y}_3^{(k)}\right]$$
$$= -\log \hat{y}_3^{(k)}$$

$(\vec{x}^{(i)}, \vec{y}^{(i)} = [1 \quad 0 \quad 0]^T)$

$(\vec{x}^{(k)}, \vec{y}^{(k)} = [0 \quad 0 \quad 1]^T)$

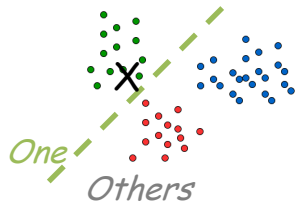


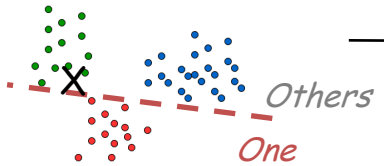
$(\vec{x}^{(j)}, \vec{y}^{(j)} = [0 \quad 1 \quad 0]^T)$

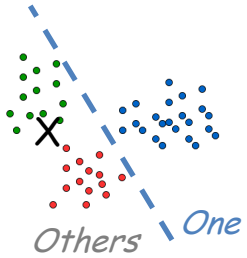
Related Properties:

$P(y = 1\|\vec{x}, \vec{w}, b) \in (0,1)$	$P(y = 1\|\vec{x}, \vec{w}, b) \to 1^-$
$\log P(y = 1\|\vec{x}, \vec{w}, b) \in (-\infty, 0)$	$\log P(y = 1\|\vec{x}, \vec{w}, b) \to 0^-$
$-\log P(y = 1\|\vec{x}, \vec{w}, b) \in (0, +\infty)$	$-\log P(y = 1\|\vec{x}, \vec{w}, b) \to 0^+$

MCCE: Calculation Example







$z_1 = 1$

$z_2 = -1$

$z_3 = -2$

$$S(z^{(j)}) = \frac{e^{z^{(j)}}}{\sum_{k=1}^{3} e^{z^{(k)}}}$$

$\hat{y}_1 = P(y_1 = 1|\vec{x}) = 0.844$

$\hat{y}_2 = P(y_2 = 1|\vec{x}) = 0.114$

$\hat{y}_3 = P(y_3 = 1|\vec{x}) = 0.042$

$$\vec{y} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \qquad \hat{\boldsymbol{Y}} = \begin{bmatrix} 0.844 \\ 0.114 \\ 0.042 \end{bmatrix}$$

Ground Truth (One-Hot Encoded Label)

Predicted

$$L_{MCCE}(\vec{y}, \hat{\boldsymbol{Y}}) = -\sum_{i=1}^{3} y_i \log \hat{y}_i$$

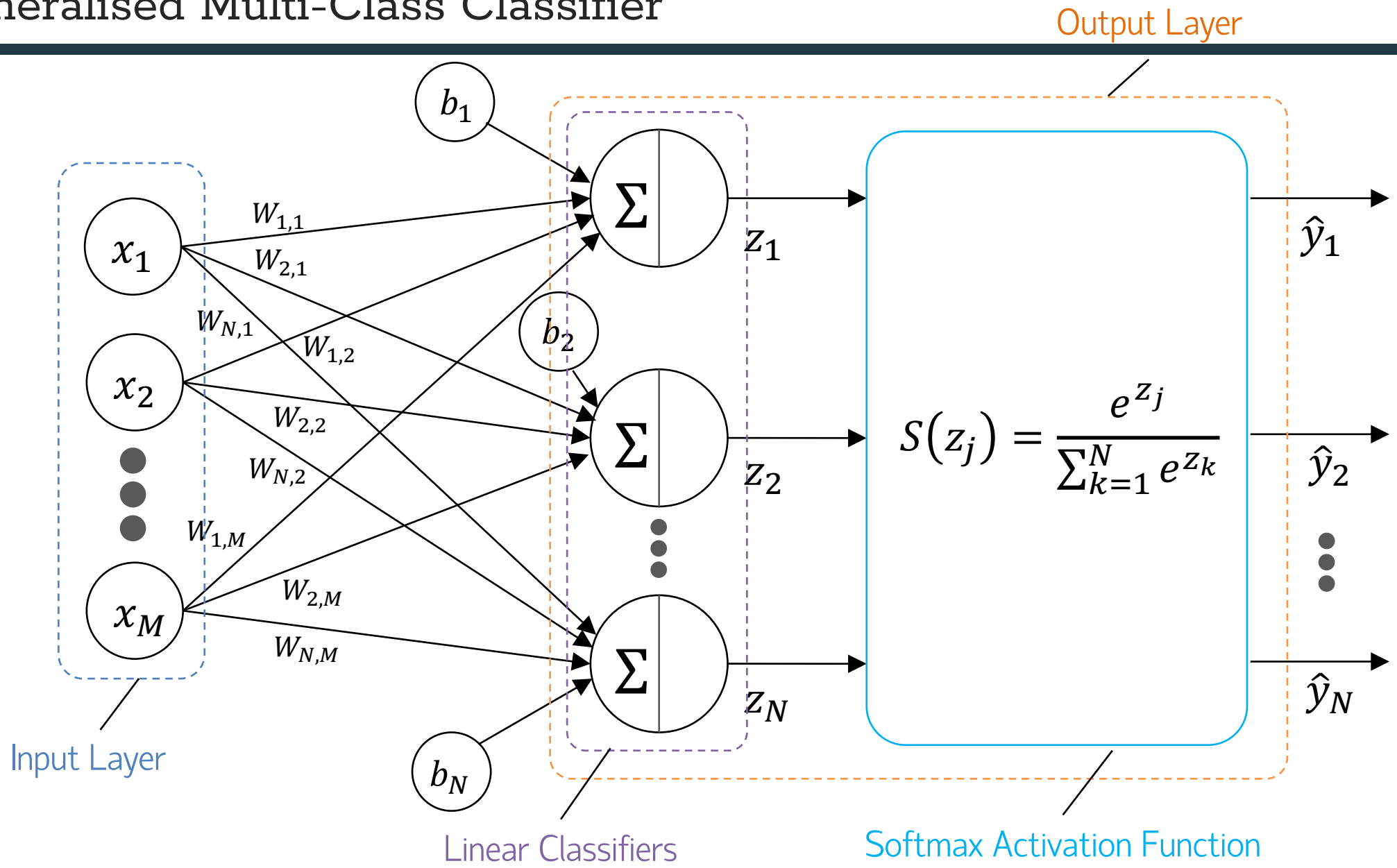
$$= -[1 \times \log 0.844 + 0 \times \log 0.114 + 0 \times 0.042]$$

$$= -[-0.17 + 0 + 0] = 0.17$$

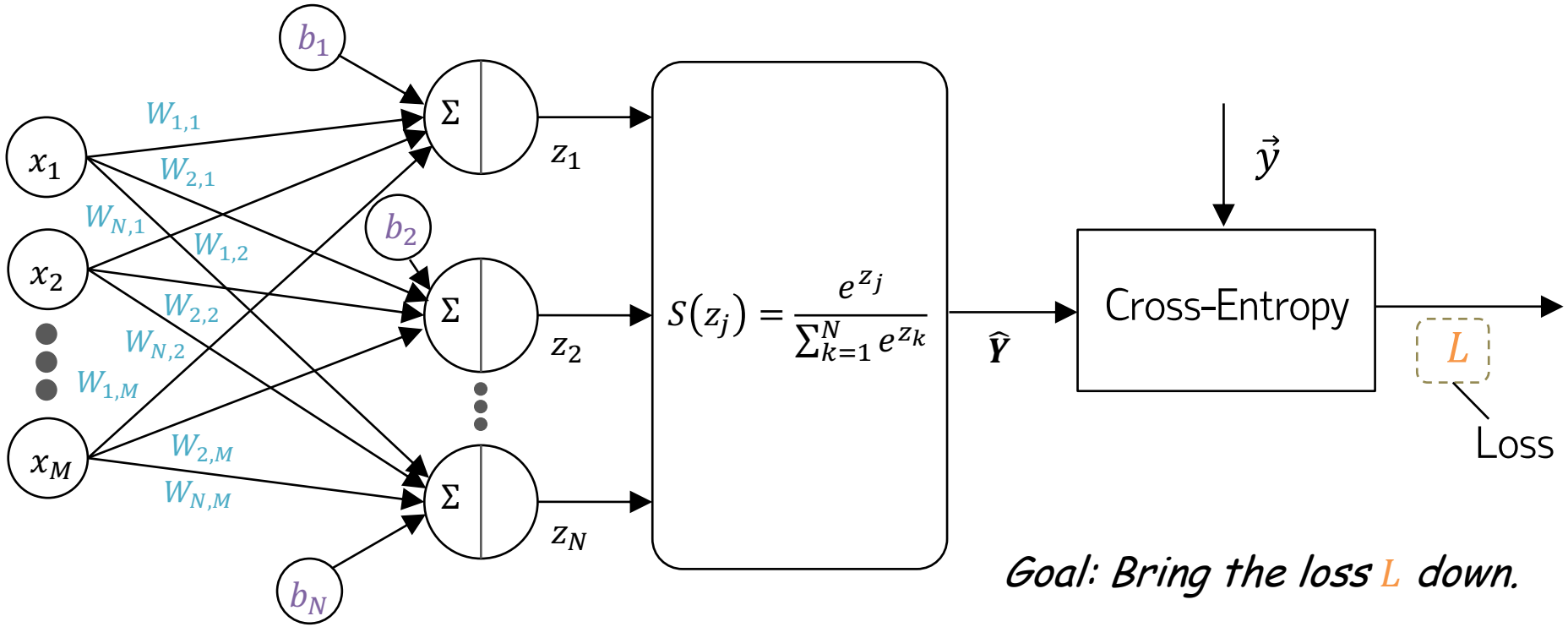
When $y_i = 1$, if $\hat{y}_i \to 1$ then $-y_i \log \hat{y}_i \to 0$ (i.e. low entropy). In reverse, if $\hat{y}_i \to 0$ then $-y_i \log \hat{y}_i \to \infty$ (i.e. high entropy).

When $y_i = 0$, $-y_i \log \hat{y}_i = -0 \times \log \hat{y}_i = 0$ regardless what $\hat{y}_i$ is. Hence, not adding to calculation.

Generalised Multi-Class Classifier



Loss Minimisation



Goal: Bring the loss L down.

Q: How will L be influenced by $W_{i,j}$ and b_i?

A: Gradients, i.e. $\frac{\partial L}{\partial W_{i,j}}$ and $\frac{\partial L}{\partial b_{i,j}}$.

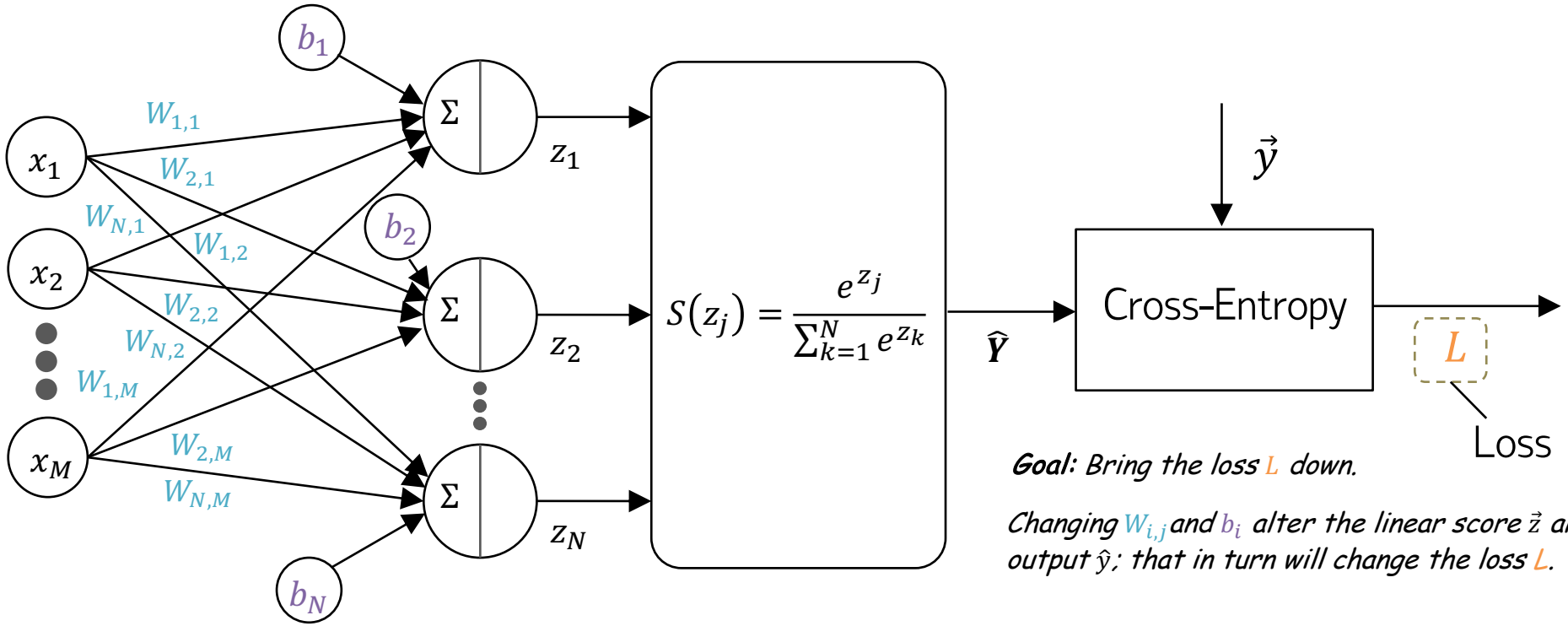
Forward Path:

$$\hat{\boldsymbol{Y}} = S\left(\boldsymbol{W} \times \vec{x} + \vec{b}\right)$$

Cross Entropy:

$$L\left(\boldsymbol{W}, \vec{b}\right) = -\frac{1}{B}\sum_{k=1}^{B} L_{MCCE}\left(\vec{y}^{(k)}, \hat{\boldsymbol{Y}}^{(i)}\right) \quad \text{where} \quad L_{MCCE}\left(\vec{y}^{(k)}, \hat{\boldsymbol{Y}}^{(i)}\right) = -\sum_{i=1}^{N} y_i \log \hat{y}_i$$

Back Propagation



Goal: *Bring the loss L down.*

Changing $W_{i,j}$ and b_i alter the linear score $\vec{z}$ and consequently changing the output $\hat{y}$; that in turn will change the loss L.

Q: How to see the changes?

A: Gradients. Chain-Rule: "Loss → Output → Linear Score → Weight/Bias"

$$\frac{\partial L}{\partial W_{i,j}} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_i} \frac{\partial z_i}{\partial W_{i,j}} \quad \text{and} \quad \frac{\partial L}{\partial b_i} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_i} \frac{\partial z_i}{\partial b_i}.$$

L is indirectly affected by $W_{i,j}$ and b_i.

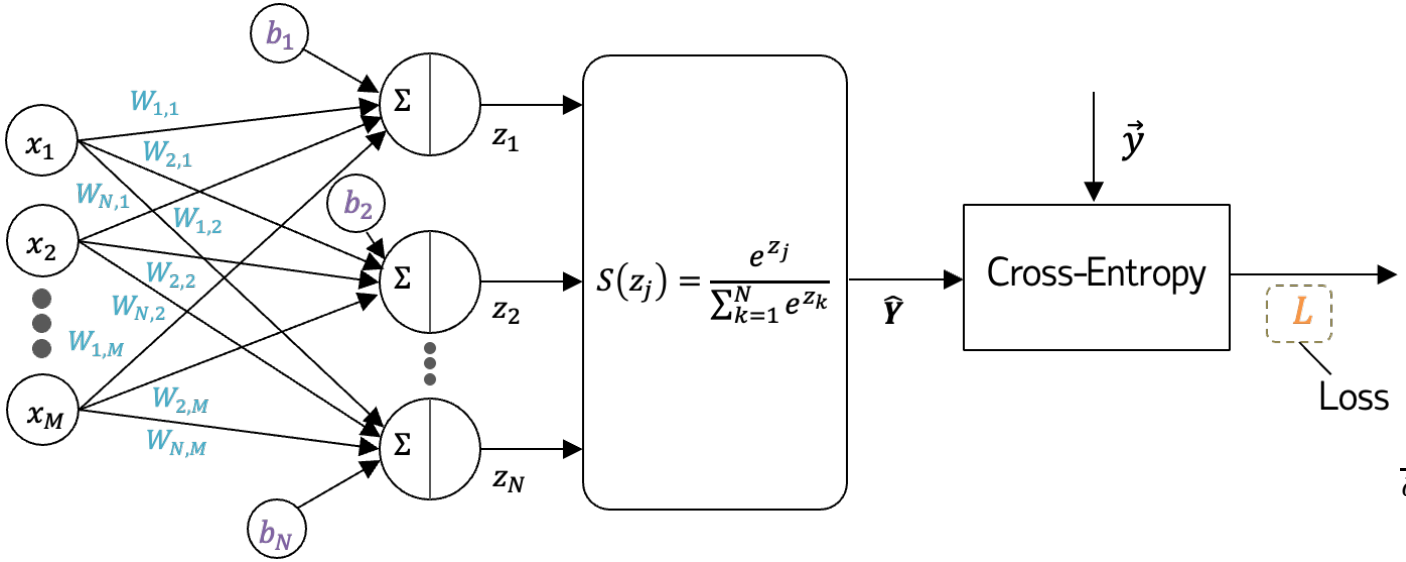
"How is $\hat{y}$ affecting L?" "How is $W_{i,j}$ affecting z_i?" "How is z_i affecting $\hat{y}$?"

$$\frac{\partial L}{\partial W_{i,j}} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_i} \frac{\partial z_i}{\partial W_{i,j}}$$

"How is $\hat{y}$ affecting L?" "How is b_i affecting z_i?" "How is z_i affecting $\hat{y}$?"

$$\frac{\partial L}{\partial b_i} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_i} \frac{\partial z_i}{\partial b_i}$$

Gradient Descent



Forward Path:

$$\hat{\boldsymbol{Y}} = S(\boldsymbol{W} \times \vec{x} + \vec{b})$$

Cross Entropy:

$$L(\boldsymbol{W}, \vec{b}) = -\frac{1}{B}\sum_{k=1}^{B} L_{MCCE}(\vec{y}^{(k)}, \hat{\boldsymbol{Y}}^{(i)})$$

where

$$L_{MCCE}(\vec{y}^{(k)}, \hat{\boldsymbol{Y}}^{(i)}) = -\sum_{i=1}^{N} y_i \log \hat{y}_i$$

"We compute $\frac{\partial L}{\partial z_I}$ once, then reuse the value to obtain $\frac{\partial L}{\partial W_{i,j}}$ and $\frac{\partial L}{\partial b_i}$ for all weights and biases."

$$\frac{\partial L}{\partial W_{i,j}} = \frac{\partial L}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial z_i}\frac{\partial z_i}{\partial W_{i,j}}$$

$$= \left(\sum_{k=1}^{N} \frac{\partial L}{\partial \hat{y}_k}\frac{\partial \hat{y}_k}{\partial z_i}\right)\frac{\partial z_i}{\partial W_{i,j}}$$

$$= \left(\sum_{k=1}^{N} \left(-\frac{y_k}{\hat{y}_k}\right)\hat{y}_k(\delta_{k,i} - \hat{y}_i)\right)x_j$$

$$= (\hat{y}_i - y_i)x_j$$

$$\frac{\partial L}{\partial b_i} = \frac{\partial L}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial z_i}\frac{\partial z_i}{\partial b_{i,}}$$

$$= \left(\sum_{k=1}^{N} \frac{\partial L}{\partial \hat{y}_k}\frac{\partial \hat{y}_k}{\partial z_i}\right)\frac{\partial z_i}{\partial b_{i,}}$$

$$= \left(\sum_{k=1}^{N} \left(-\frac{y_k}{\hat{y}_k}\right)\hat{y}_k(\delta_{k,i} - \hat{y}_i)\right)$$

$$= (\hat{y}_i - y_i)$$

Update: Step against the Slope

$$W_{i,j}^{(t+1)} \leftarrow W_{i,j}^{(t)} - \eta \frac{\partial L}{\partial W_{i,j}}(W_{i,j}^{(t)})$$

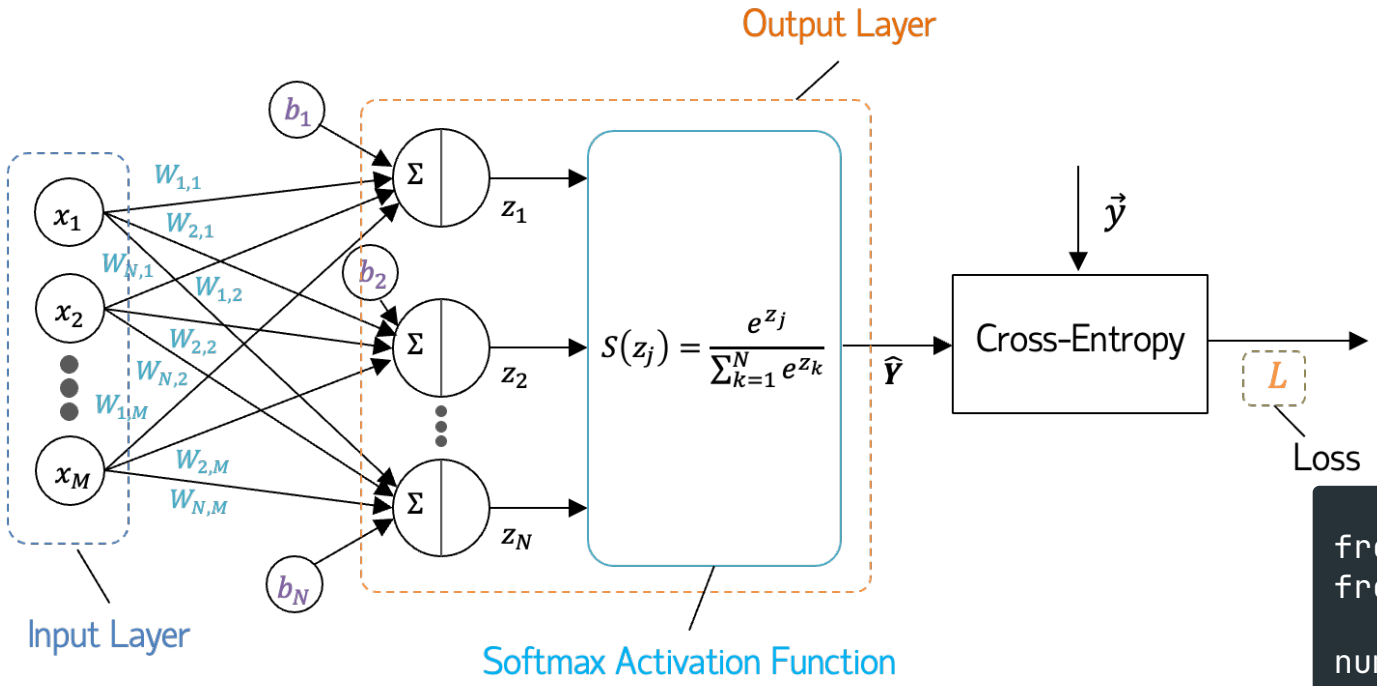
$$\leftarrow W_{i,j}^{(t)} - \eta \times \left(\frac{1}{B}\sum_{k=1}^{B}\left(\hat{y}_i^{(k)} - y_i^{(k)}\right)x_j^{(k)}\right)$$

$$b_i^{(t+1)} \leftarrow b_i^{(t)} - \eta \frac{\partial L}{\partial b_i b}(b_i^{(t)})$$

$$\leftarrow b_i^{(t)} - \eta \times \left(\frac{1}{B}\sum_{k=1}^{B}\left(\hat{y}_i^{(k)} - y_i^{(k)}\right)\right)$$

Function	Derivative $\left(\frac{\partial}{\partial z_i}\right)$
$S(z_i)$	$\sigma(z)(1-\sigma(z))$
$\log \hat{y}_i$	$\delta_{j,i} - \hat{y}_i$
$-\sum_{k=1}^{N} k_k \log \hat{y}_k$	$\hat{y}_i - y_i$

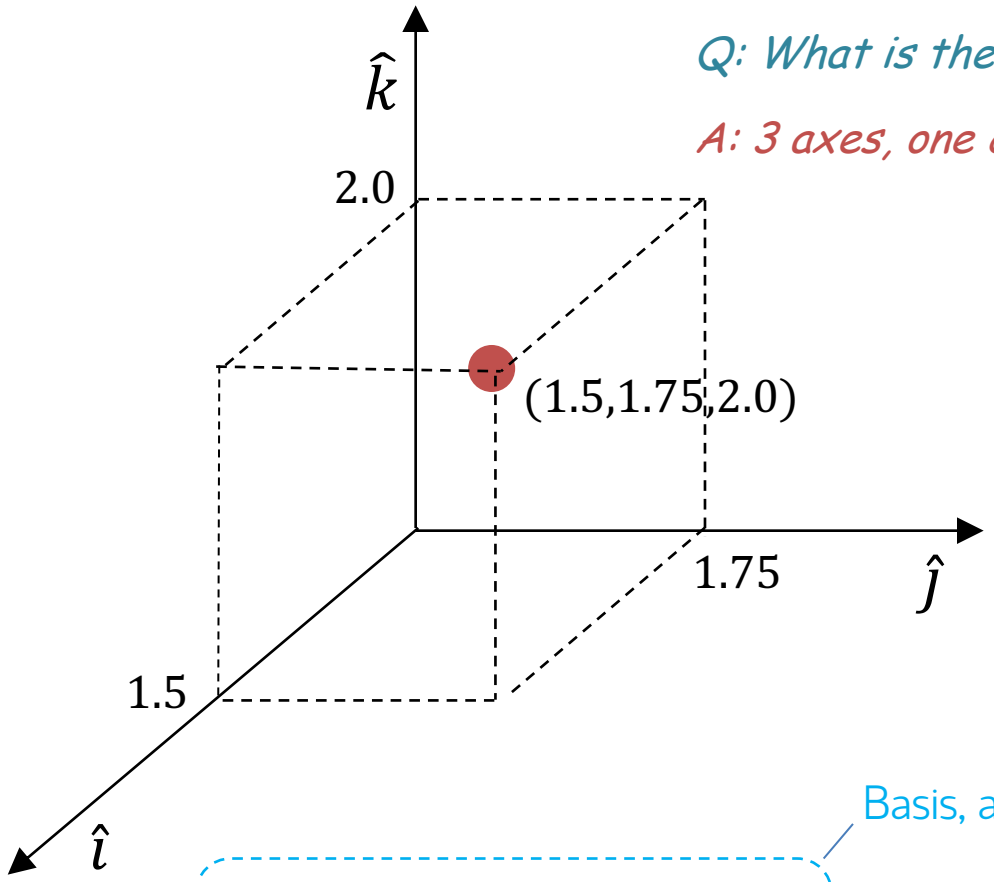
Tensorflow Code Snippet




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

num_inputs = M          # input dimension
num_outputs = N         # input dimension

model = Sequential([
    Dense(num_outputs,
          activation="softmax”,
          input_shape=(num_inputs,))
])

model.compile(optimizer=”sgd",
              loss=”categorical_crossentropy",
              metrics=["accuracy"])
```

TensorFlow

Metric Space (3D)



Q: What is the basis (or features in ML) required to represent (1.5, 1.75, 2.0)?

A: 3 axes, one cannot be derived from the others.

Basis, aka ML Features

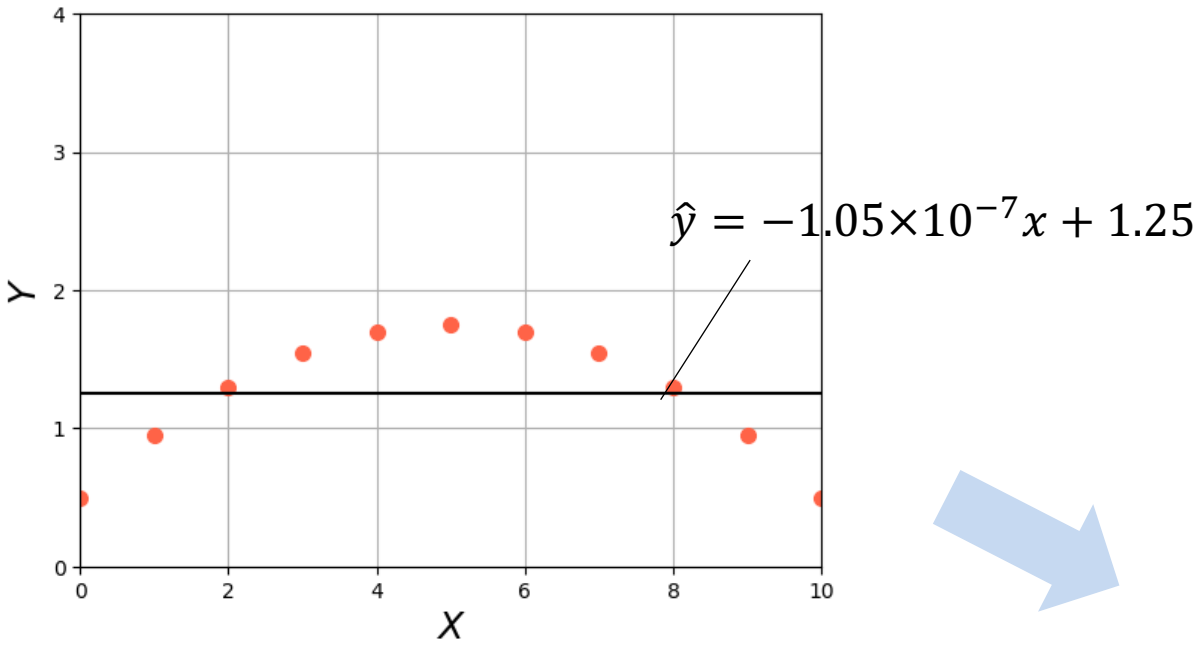
$$\hat{\imath} = \begin{bmatrix}1\\0\\0\end{bmatrix} \quad \hat{\jmath} = \begin{bmatrix}0\\1\\0\end{bmatrix} \quad \hat{k} = \begin{bmatrix}0\\0\\1\end{bmatrix}$$

$$\vec{x} = 1.5\times\hat{\imath} + 1.75\times\hat{\jmath} + 2.0\times\hat{k}$$

$$= 1.5\times\begin{bmatrix}1\\0\\0\end{bmatrix} + 1.75\times\begin{bmatrix}0\\1\\0\end{bmatrix} + 2.0\times\begin{bmatrix}0\\0\\1\end{bmatrix}$$

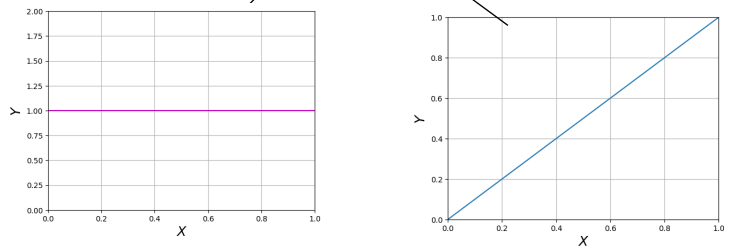
$$= \begin{bmatrix}1.5\\1.75\\2.0\end{bmatrix}$$

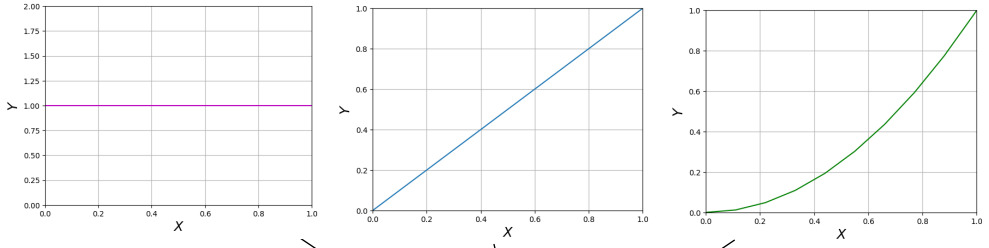
Functional Space: Polynomial Regression



Features

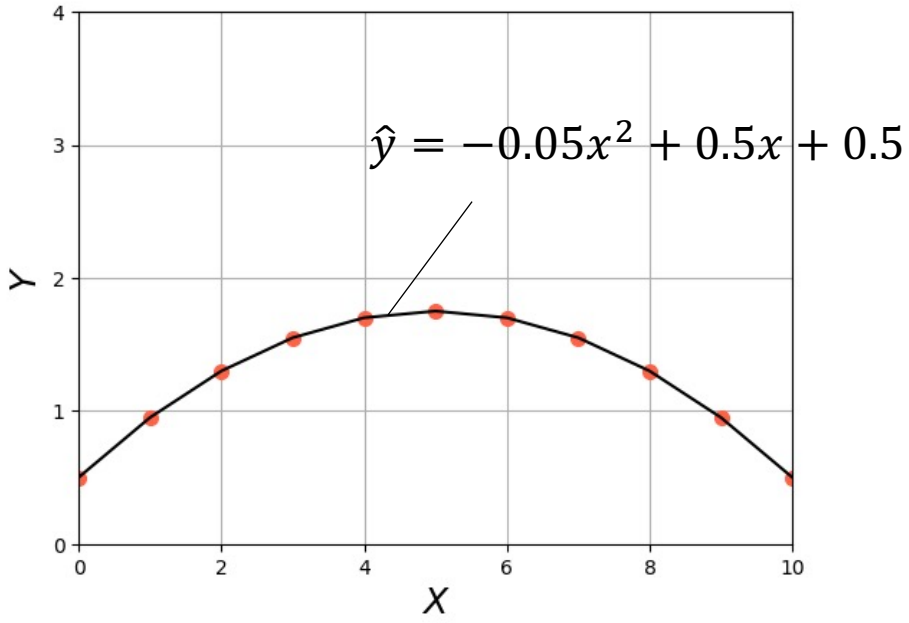
Basis: $[1 \quad x]^T$



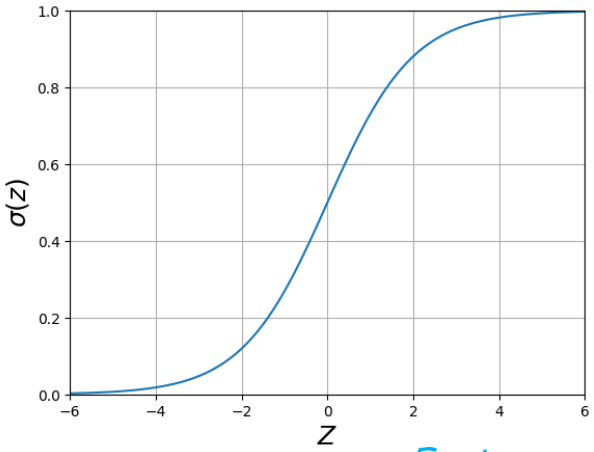


Basis: $[1 \quad x \quad x^2]^T$

Features



Sigmoidal Basis Function



Sigmoid Function:

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

where $z = w \times x + b$ Is a linear score.

$\sigma(z)$ is parametrised by w and b

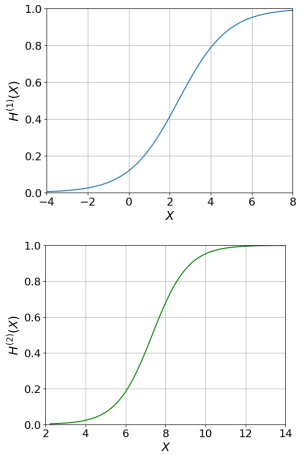
Features

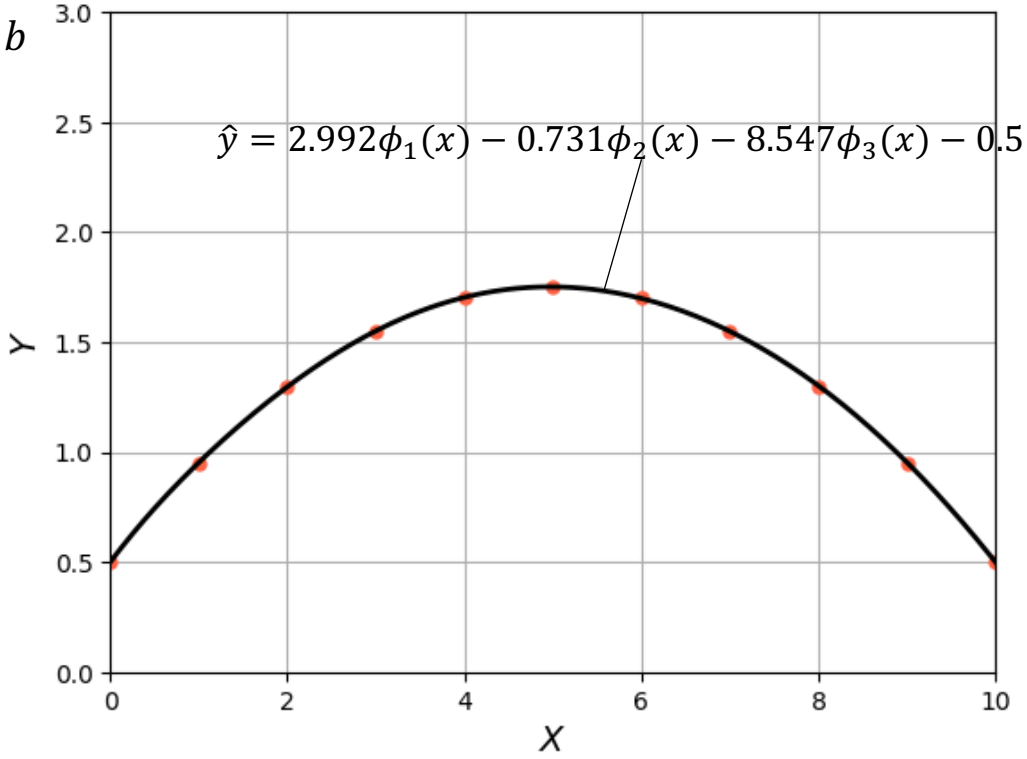
Basis: $[1 \quad \phi_1(x) \quad \phi_2(x) \quad \phi_3(x)\ ]^T$

$$\phi_1(x) = \frac{1}{1 + e^{-(0.517x - 0.369)}}$$

$$\phi_2(x) = \frac{1}{1 + e^{-(1.890x - 1.578)}}$$

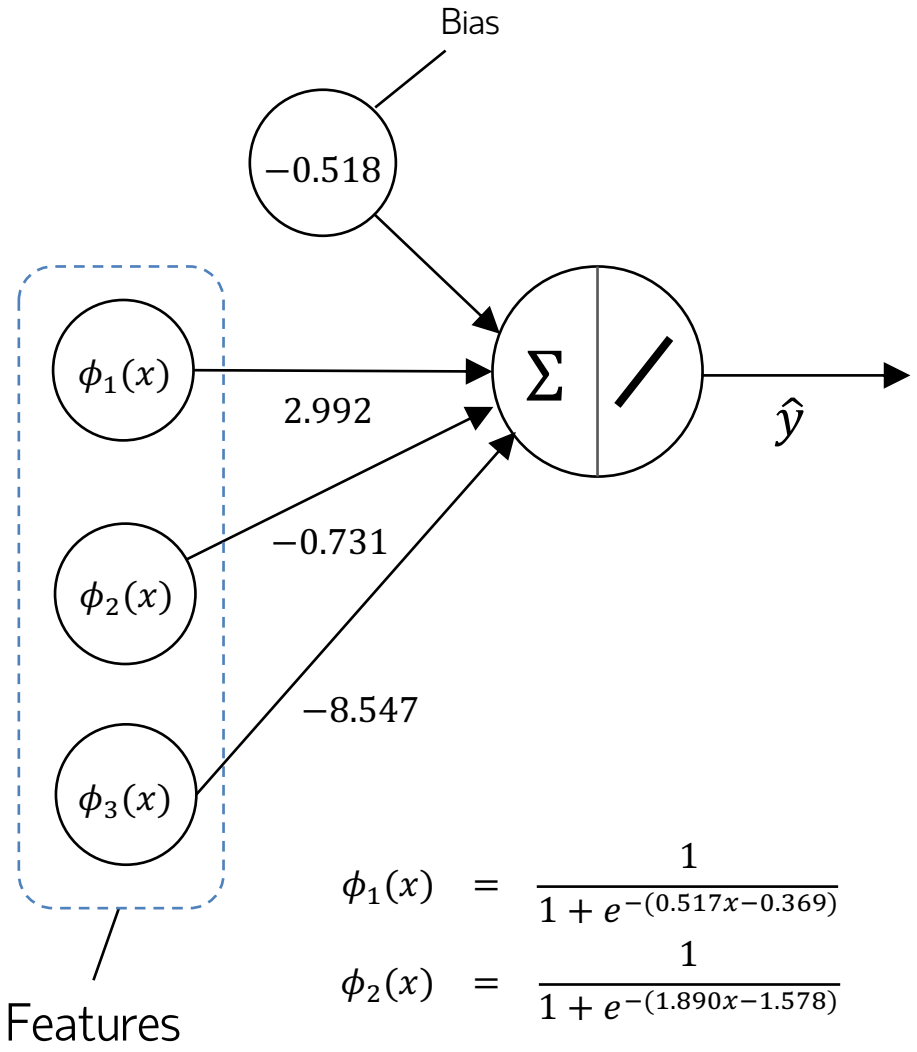
$$\phi_3(x) = \frac{1}{1 + e^{-(0.344x - 4.660)}}$$





With many of $\sigma(z)$s, we can capture any underlying complexities.

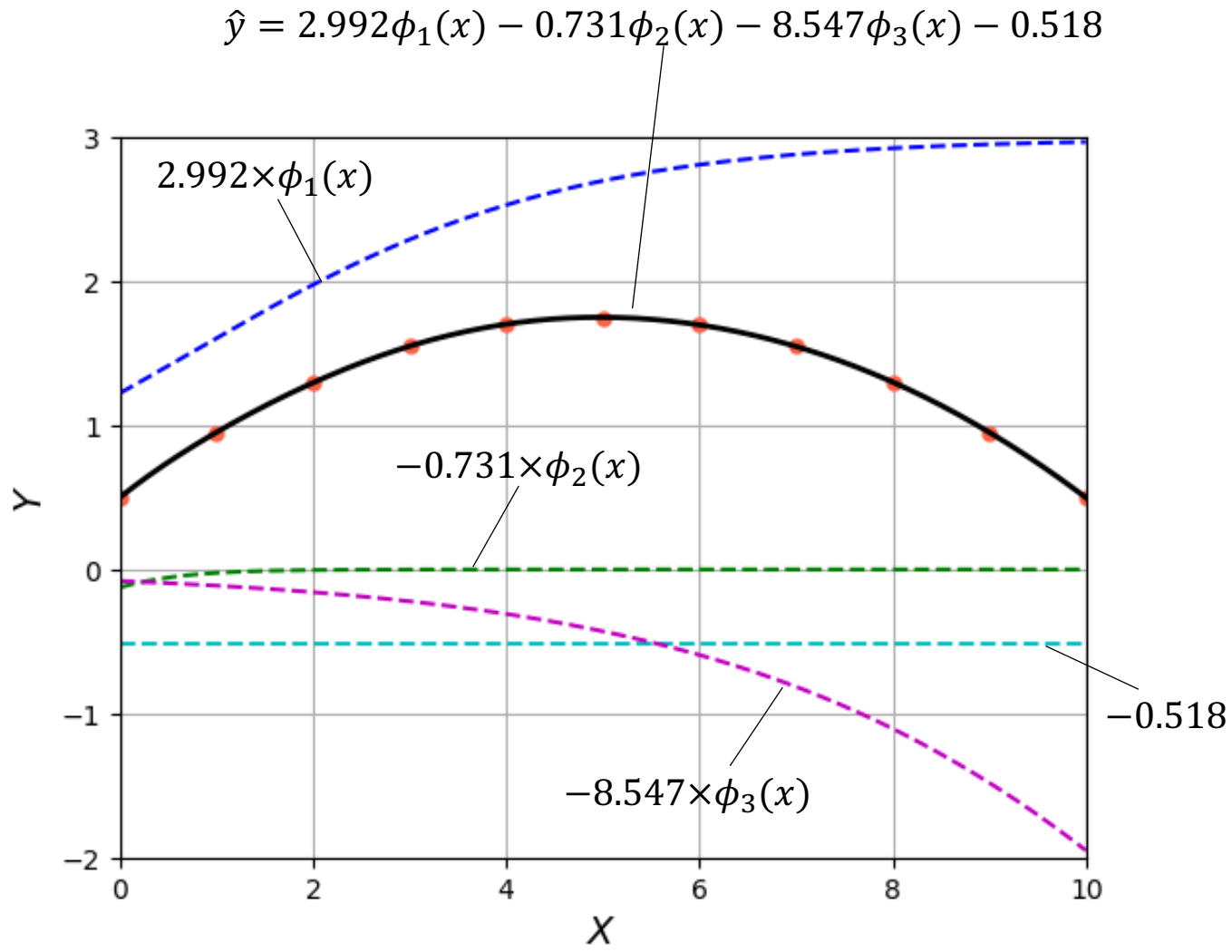
Sigmoidal Basis Function (cont.)



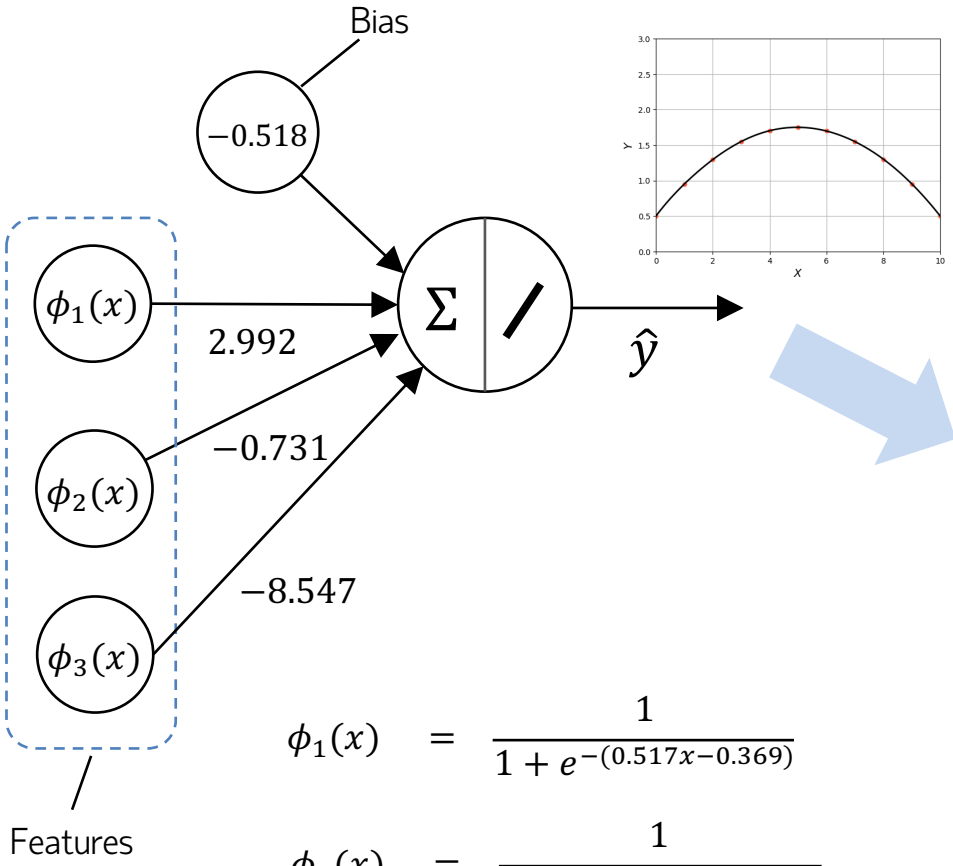
$$\phi_1(x) = \frac{1}{1+e^{-(0.517x-0.369)}}$$

$$\phi_2(x) = \frac{1}{1+e^{-(1.890x-1.578)}}$$

$$\phi_3(x) = \frac{1}{1+e^{-(0.344x-4.660)}}$$



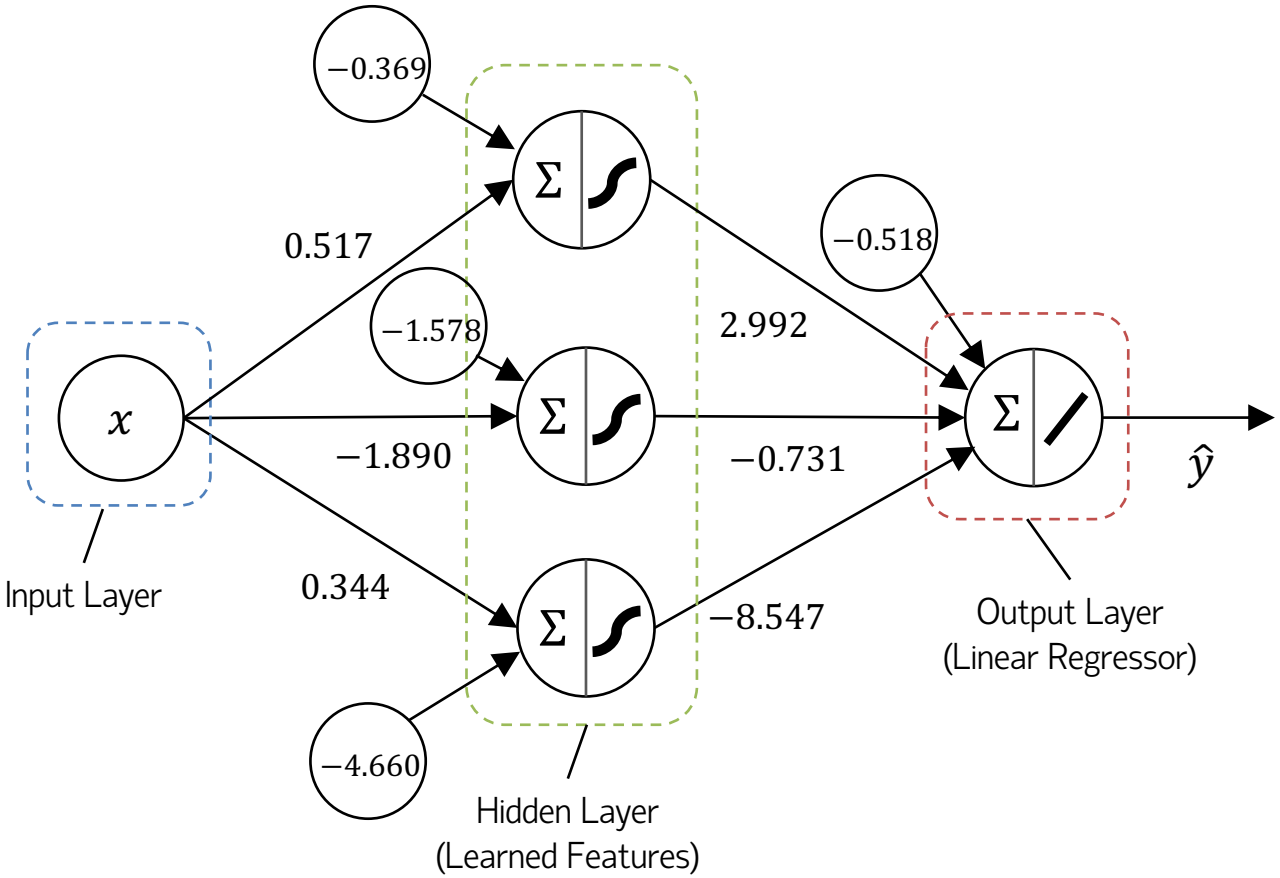
Hidden Layer



$$\phi_1(x) = \frac{1}{1+e^{-(0.517x-0.369)}}$$

$$\phi_2(x) = \frac{1}{1+e^{-(-1.890x-1.578)}}$$

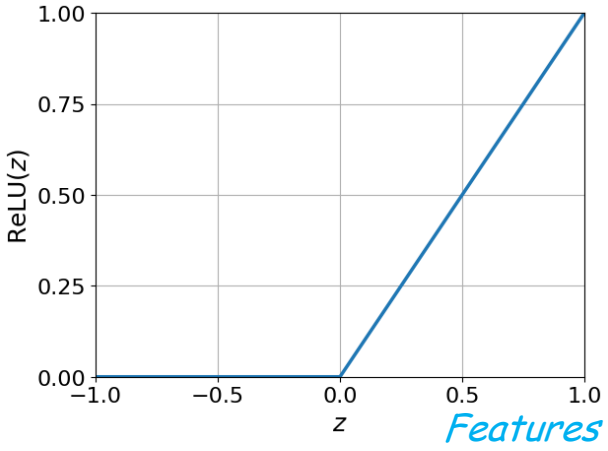
$$\phi_3(x) = \frac{1}{1+e^{-(0.344x-4.660)}}$$



Feature ↔ Neuron with Sigmoid Activation Function

Our features share a common structure, parametrised by w and b.

Rectified Linear Unit (ReLU) Basis Function



Features

Rectified Linear Unit Function:

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

where $\text{z} = w \times x + b$ Is a linear score.

$\text{ReLU}(z)$ is parametrised by w and b

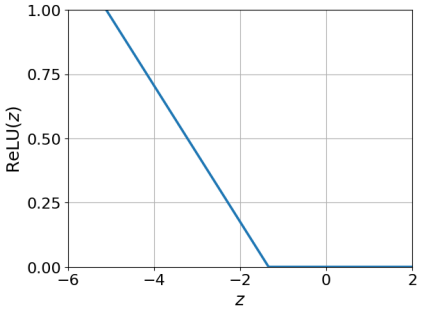
Basis: $[1 \quad \phi_1(x) \quad \cdots \quad \phi_{14}(x)]^T$

$$\vec{w}_H = [-0.265 \quad -0.507 \quad \cdots \quad 0.529]^T$$
$$\vec{b}_H = [-0.355 \quad -0.123 \quad \cdots \quad 0.151]^T$$

$$\phi_i(x) = \begin{cases} w_i^{(L)} \times x + b_i^{(L)} & w_i^{(L)} \times x + b_i^{(L)} \geq 0 \\ 0 & w_i^{(L)} \times x + b_i^{(L)} < 0 \end{cases}$$

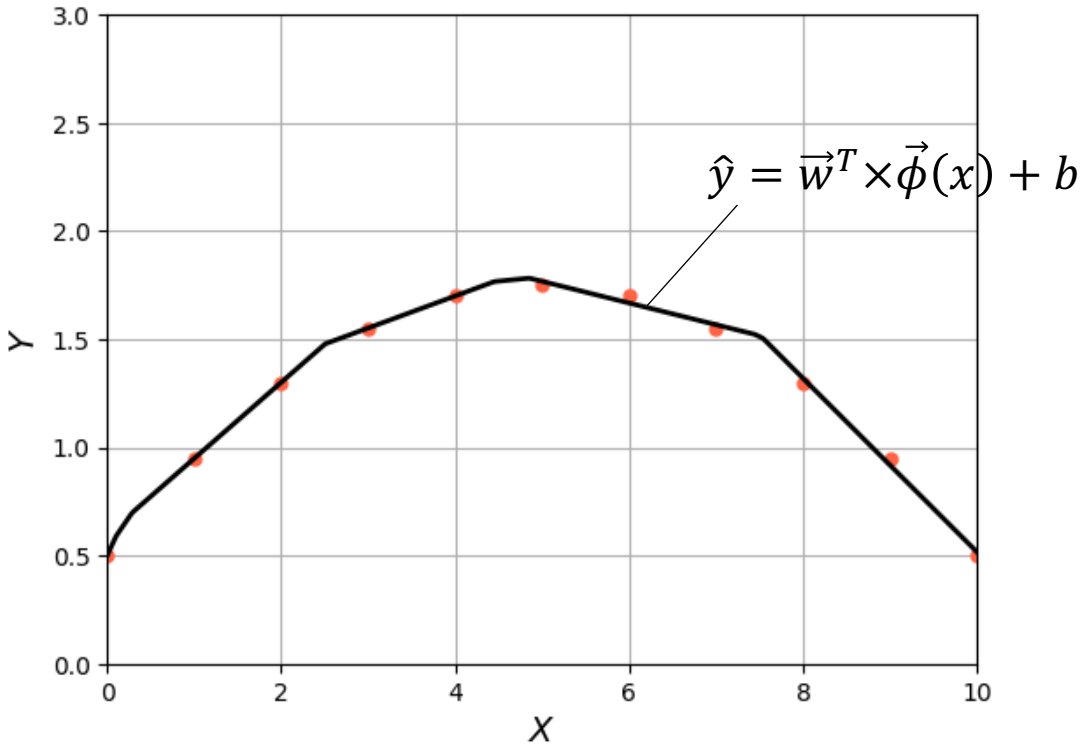
Example:

$$\phi_1(x) = \begin{cases} -0.265x - 0.355 & -0.265x - 0.355 \geq 0 \\ 0 & -0.265x - 0.355 < 0 \end{cases}$$



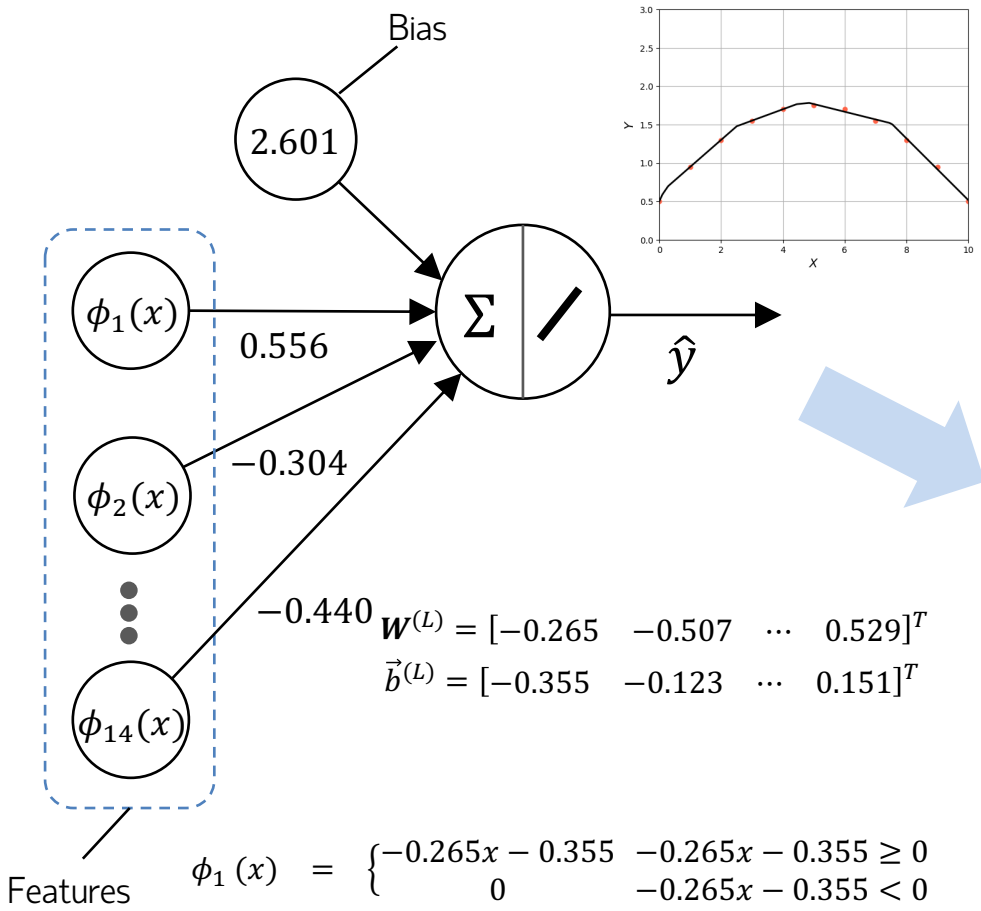
$$\vec{w} = [0.556 \quad -0.304 \quad \cdots \quad -0.440]^T$$

$$b = 2.601$$



With many of $\text{ReLU}(z)$*s, we can capture any underlying complexities.*

Hidden Layer (with ReLU Function)

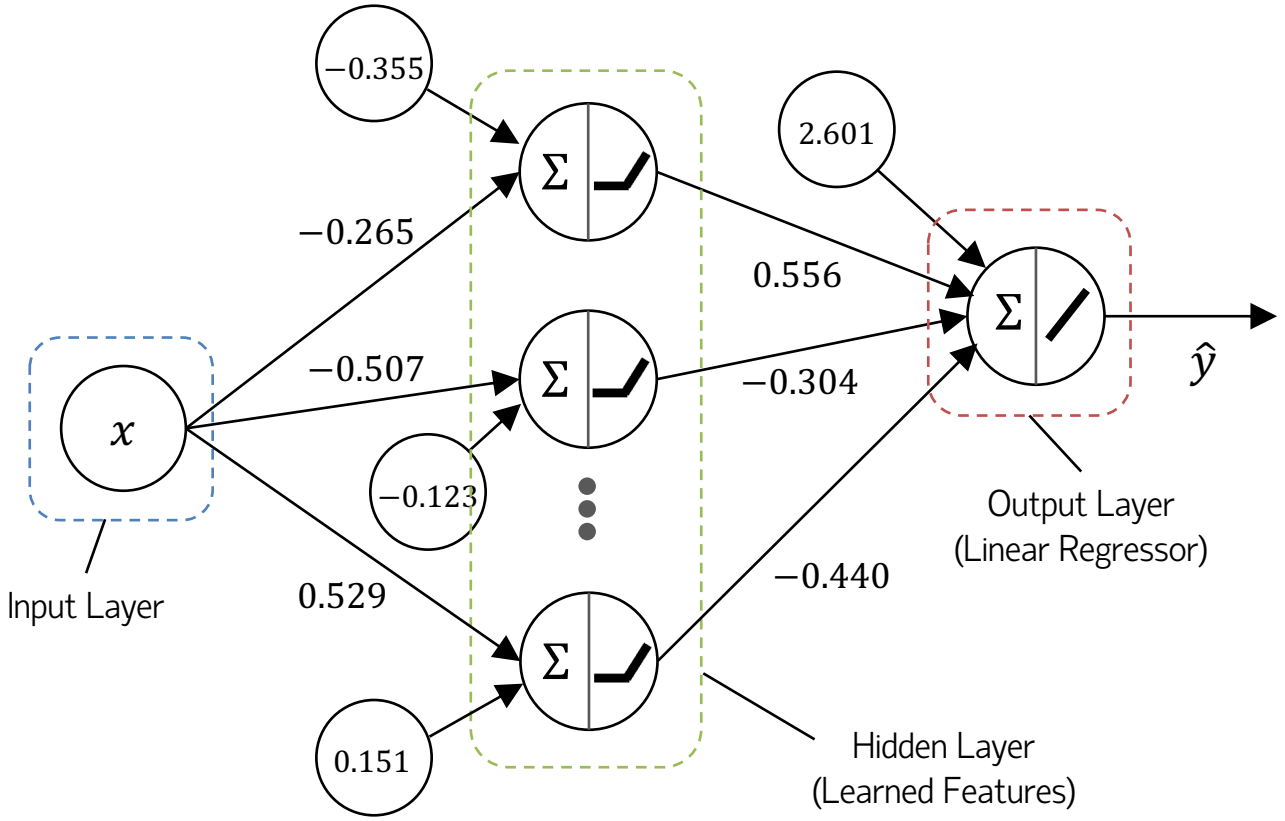


$$\boldsymbol{W}^{(L)} = [-0.265 \quad -0.507 \quad \cdots \quad 0.529]^T$$

$$\vec{b}^{(L)} = [-0.355 \quad -0.123 \quad \cdots \quad 0.151]^T$$

$$\phi_1(x) = \begin{cases} -0.265x - 0.355 & -0.265x - 0.355 \geq 0 \\ 0 & -0.265x - 0.355 < 0 \end{cases}$$

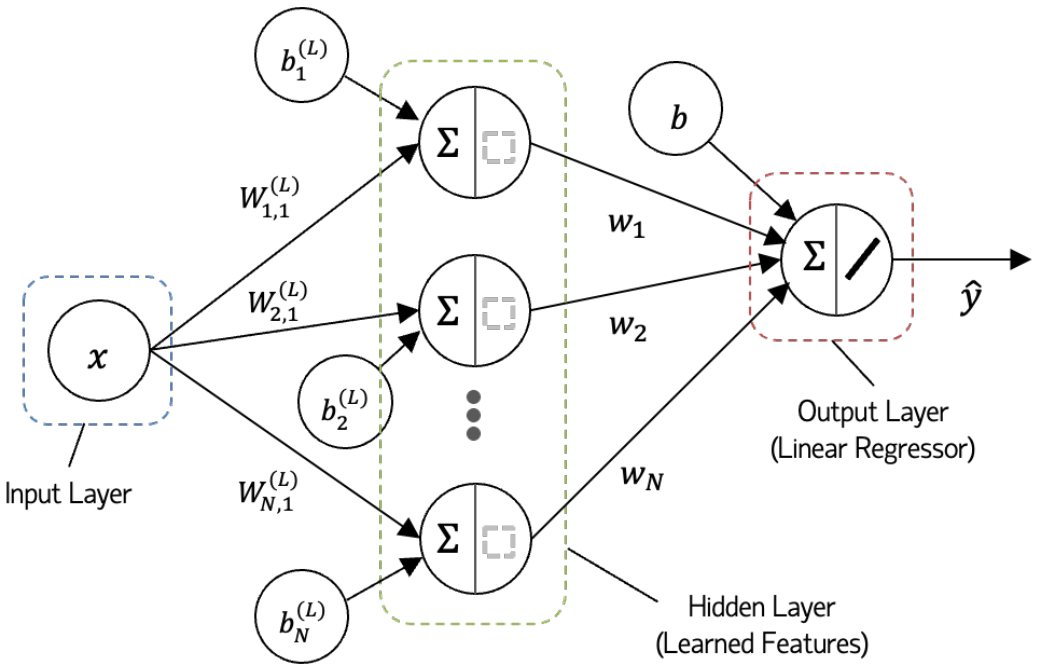
$$\phi_{14}(x) = \begin{cases} 0.529x + 0.151 & 0.529x + 0.151 \geq 0 \\ 0 & 0.529x + 0.151 < 0 \end{cases}$$

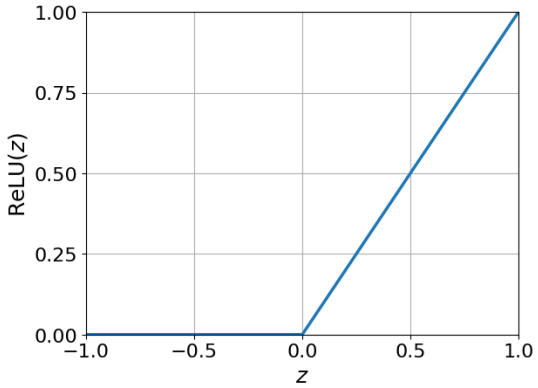


Feature ↔ Neuron with ReLU Activation Function

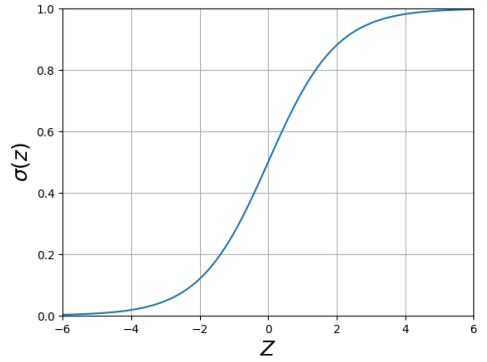
Our features share a common ReLU structure, parametrised by w and b.

Bases for Hidden Layer

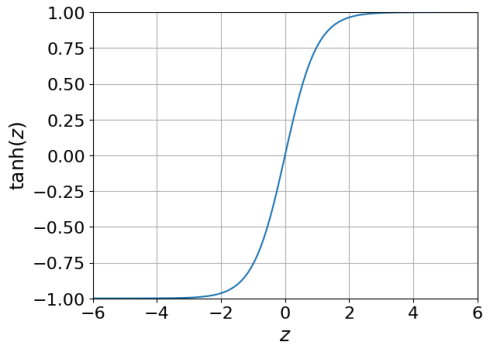




ReLU (and its variants) is the default for modern feed-forward hidden layers—positive inputs pass through, negatives drop to zero—giving sparse activations, fast convergence, and gradients that survive even in very deep nets.



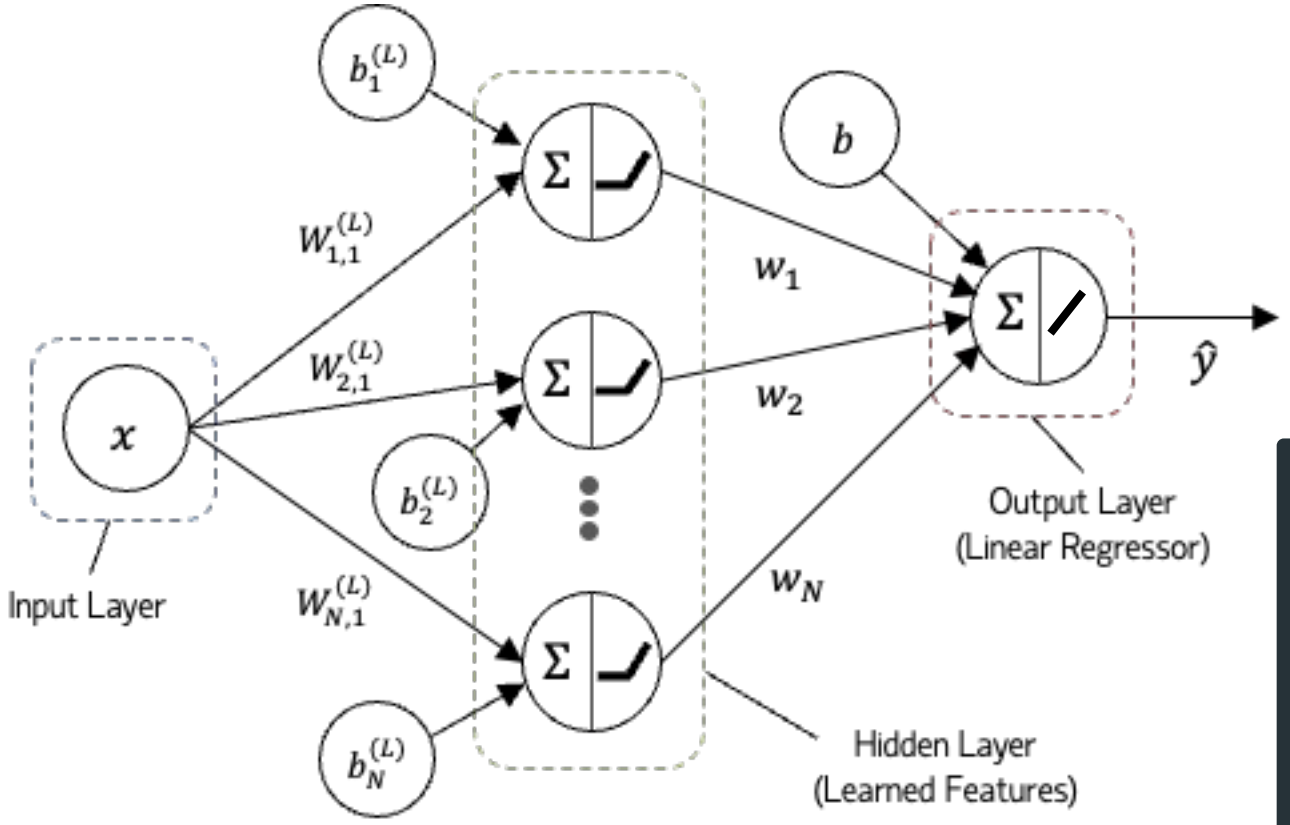
Sigmoid is mainly kept for output neurons or for gates inside LSTM/GRU cells—its (0 , 1) range turns a logit into a probability, but in deep hidden stacks it saturates and kills gradients.



Tanh remains the work-horse hidden activation in RNNs: its (−1 … 1) output keeps recurrent states zero-centred and bounded, stabilising long-sequence training, yet very deep tanh layers can still suffer from fading gradients.

Common thread: *each activation supplies a fixed "basis" curve for every neuron, and gradient descent simply slides and scales that curve via the weights* $W_{i,j}$ *and bias* b_i *to carve out the features the network needs.*

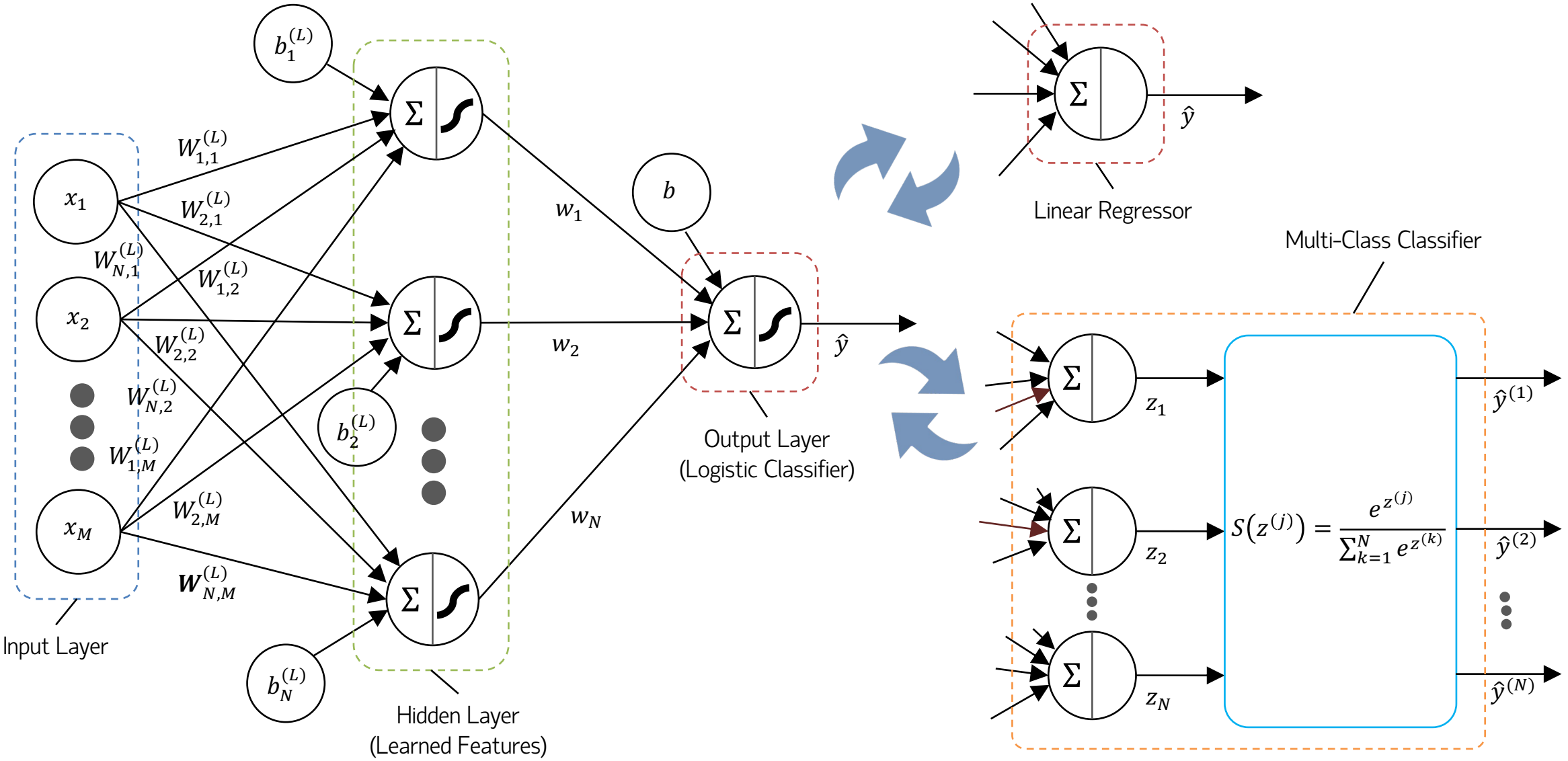
Tensorflow Code Snippet



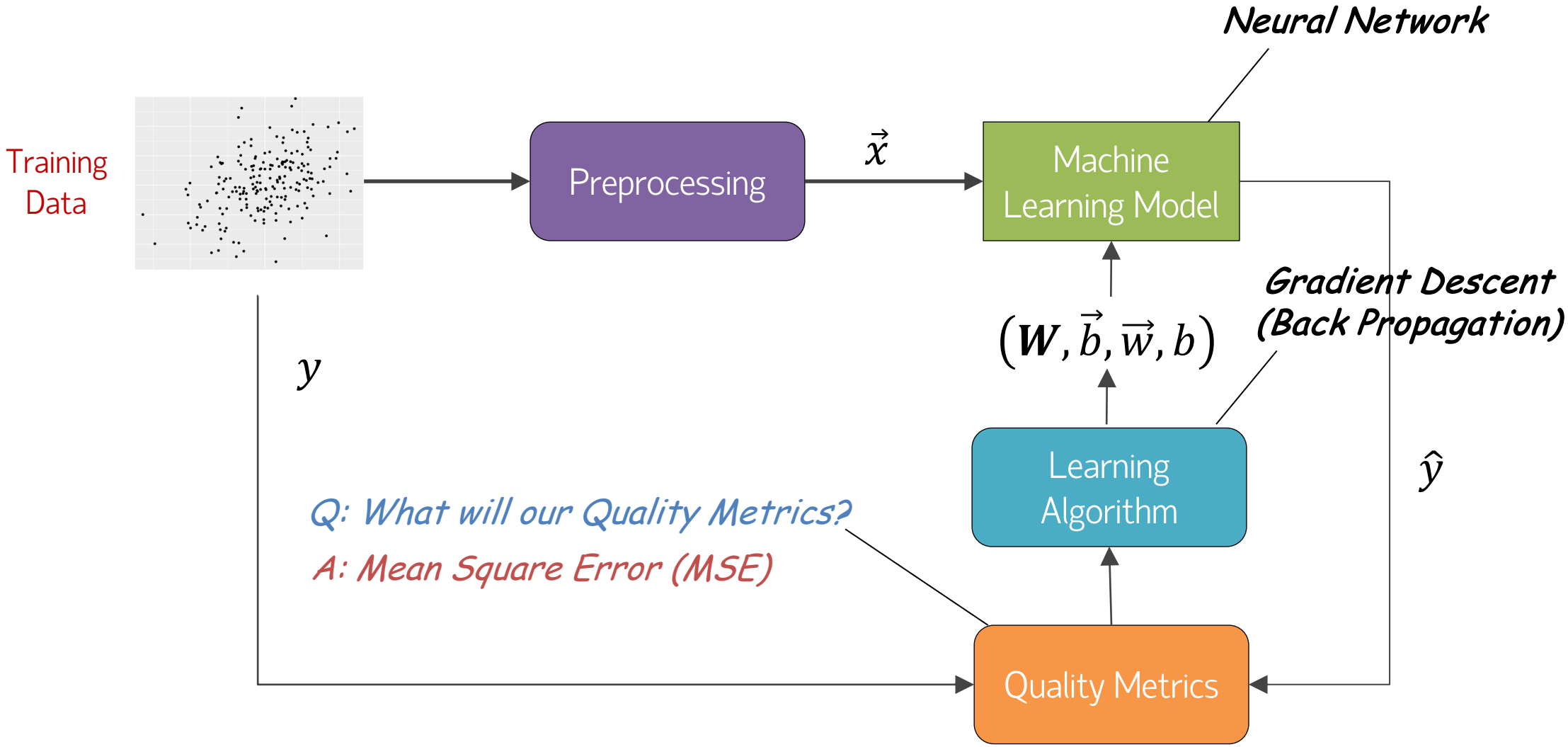

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense

num_inputs = 1          # input dimension
num_hidden_units = 14   # hidden units
num_outputs = 1         # hidden units


model = Sequential([
    Dense(num_hidden_units,
          activation="relu",
          input_shape=(num_inputs,)),

    Dense(num_outputs,
          activation="linear")
])
```

TensorFlow

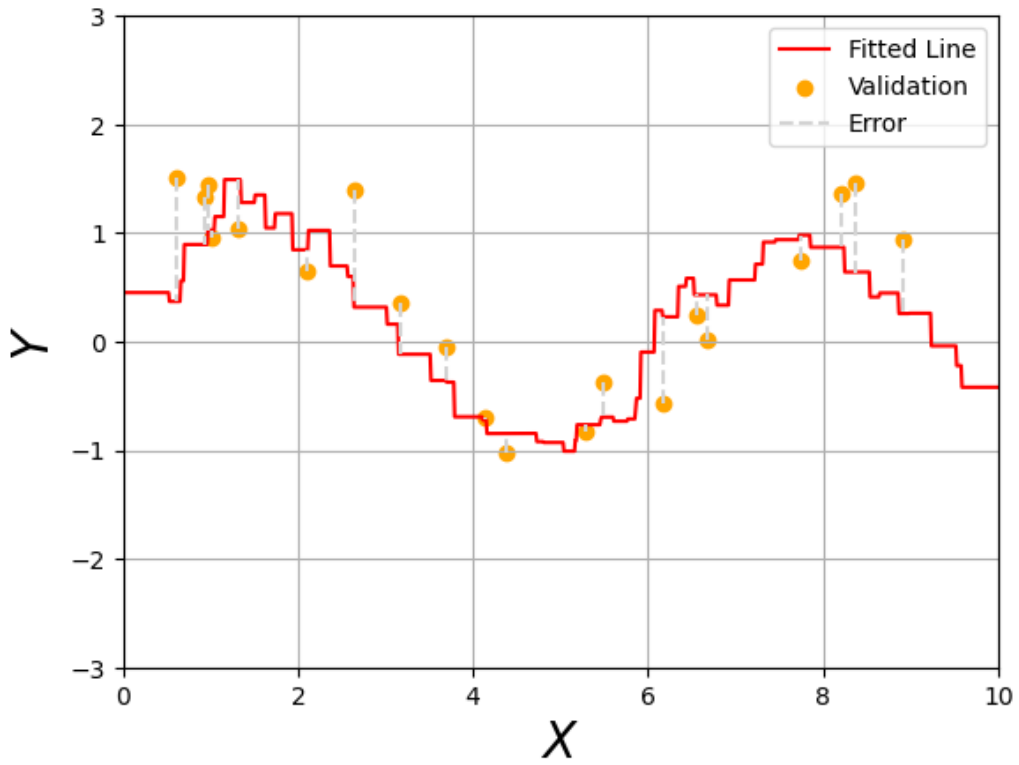
Generalisation



Workflow: Regression



Regression Line



$$\text{Error (Residue)} \quad = \quad y - \hat{y}$$

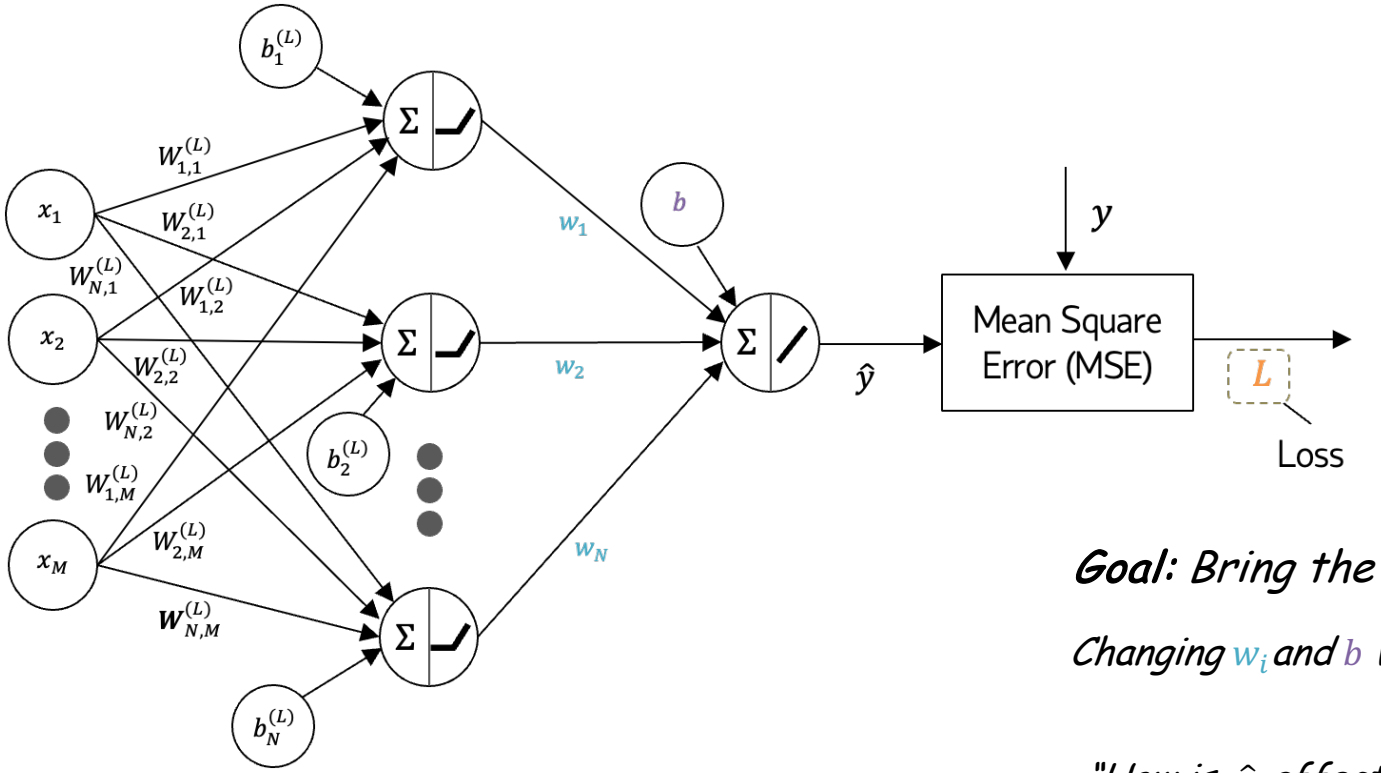
where y and $\hat{y}$ are the observed and predicted values, respectively.

Mean Squared Error (MSE):

$$\text{MSE} \quad = \quad \frac{1}{B}\sum_{k=1}^{B}\left(y^{(k)} - \hat{y}^{(k)}\right)^2$$

where B is the total number of data points.

Output Layer: Gradients



Forward Path:

$$\Phi^{(L)}(\vec{x}) = \sigma\left(\boldsymbol{W}^{(L)} \times \vec{x} + \vec{b}^{(L)}\right)$$

$$\hat{y} = \vec{w}^T \times \Phi^{(L)}(\vec{x}) + b$$

Mean Square Error:

$$L(\boldsymbol{W}, \vec{b}) = \frac{1}{B}\sum_{k=1}^{B}(y - \hat{y})^2$$

Goal: *Bring the loss L down.*

Changing w_i and b will change the out $\hat{y}$; that in turn will change the loss L.

Q: How to see the changes?

A: Gradients. Chain-Rule: "Loss → Output → Weight/Bias"

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial w_i} \quad \text{and} \quad \frac{\partial L}{\partial b} = \frac{\partial L}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial b}.$$

L is indirectly affected by w_i and b.

"How is $\hat{y}$ affecting L?"

$$\frac{\partial L}{\partial W_{i,j}} = \frac{\partial L}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial w_i}$$

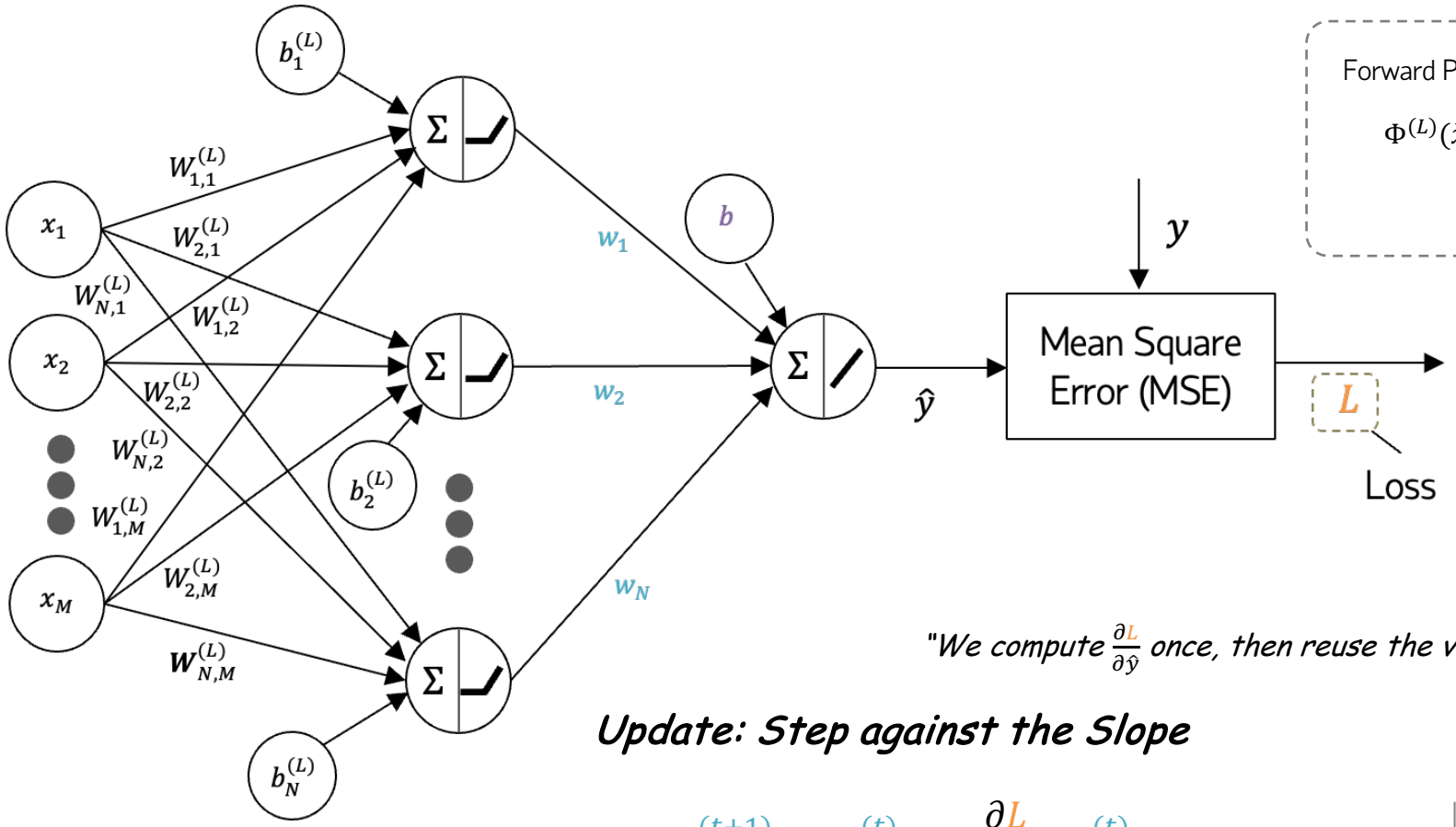
"How is w_i affecting $\hat{y}$?"

"How is b affecting $\hat{y}$?"

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial b}$$

"How is $\hat{y}$ affecting L?"

Output Layer: Gradient Descent



Forward Path:

$$\Phi^{(L)}(\vec{x}) = \sigma\left(\boldsymbol{W}^{(L)} \times \vec{x} + \vec{b}^{(L)}\right)$$

$$\hat{y} = \vec{w}^T \times \Phi^{(L)}(\vec{x}) + b$$

Mean Square Error:

$$L(\boldsymbol{W}, \vec{b}) = \frac{1}{B}\sum_{k=1}^{B}(y - \hat{y})^2$$

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial w_i} = -2(y - \hat{y})\phi_i$$

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial b} = -2(y - \hat{y})$$

"We compute $\frac{\partial L}{\partial \hat{y}}$ once, then reuse the value to obtain $\frac{\partial L}{\partial w_i}$ and $\frac{\partial L}{\partial b}$ for all weights and biases."

Update: Step against the Slope

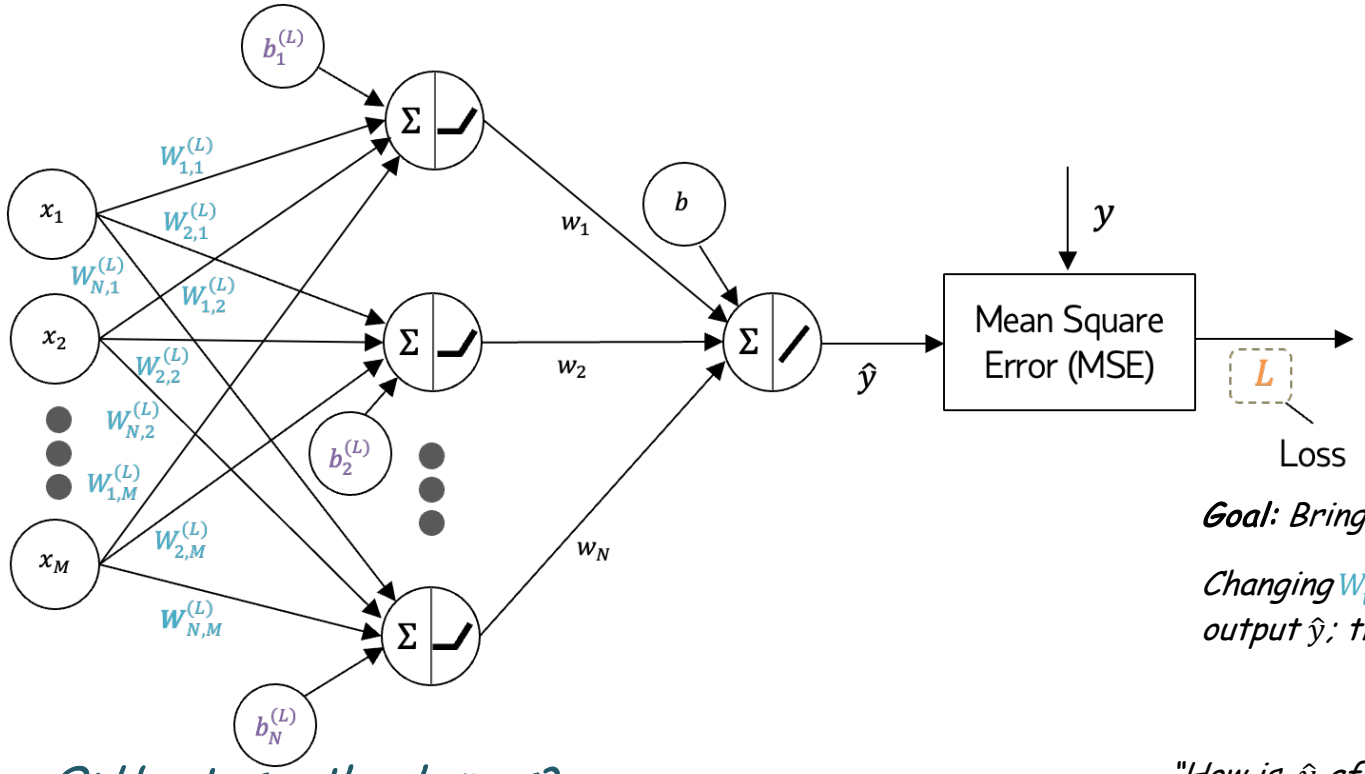
$$w_i^{(t+1)} \leftarrow w_i^{(t)} - \eta \frac{\partial L}{\partial w_i}(w_i^{(t)})$$

$$\leftarrow w_i^{(t)} - \eta \times \left(\frac{2}{B}\sum_{k=1}^{B}\left(\hat{y}^{(k)} - y^{(k)}\right)\phi_i^{(k)}\right)$$

$$b^{(t+1)} \leftarrow b^{(t)} - \eta \frac{\partial L}{\partial b}(b^{(t)})$$

$$\leftarrow b^{(t)} - \eta \times \left(\frac{2}{B}\sum_{k=1}^{B}\left(\hat{y}^{(k)} - y^{(k)}\right)\right)$$

Hidden Layer: Back Propagation



Forward Path:

$$\Phi^{(L)}(\vec{x}) = \mathrm{ReLU}\left(\boldsymbol{W}^{(L)} \times \vec{x} + \vec{b}^{(L)}\right)$$

$$\hat{y} = \vec{w}^T \times \Phi^{(L)}(\vec{x}) + b$$

Mean Square Error:

$$L(\boldsymbol{W}, \vec{b}) = \frac{1}{B}\sum_{k=1}^{B}(y - \hat{y})^2$$

Goal: *Bring the loss L down.*

Changing $W_{i,j}^{(L)}$ and $b_i^{(L)}$ alter the feature $\phi_i^{(L)}$ and consequently changing the output $\hat{y}$; that in turn will change the loss L.

Q: How to see the changes?

A: Gradients. Chain-Rule: "Loss → Output → Feature → Weight/Bias"

$$\frac{\partial L}{\partial W_{i,j}^{(L)}} = \frac{\partial L}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial \phi_i^{(L)}}\frac{\partial \phi_i^{(L)}}{\partial W_{i,j}^{(L)}} \quad \text{and} \quad \frac{\partial L}{\partial b_i^{(L)}} = \frac{\partial L}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial \phi_i^{(L)}}\frac{\partial \phi_i^{(L)}}{\partial b_i^{(L)}}.$$

L is indirectly affected by $W_{i,j}^{(L)}$ and $b_i^{(L)}$.

"How is $W_{i,j}$ affecting ϕ_i?"

"How is $\hat{y}$ affecting L?"

$$\frac{\partial L}{\partial W_{i,j}^{(L)}} = \frac{\partial L}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial \phi_i^{(L)}}\frac{\partial \phi_i^{(L)}}{\partial W_{i,j}^{(L)}}$$

"How is $\phi_i^{(L)}$ affecting $\hat{y}$?"

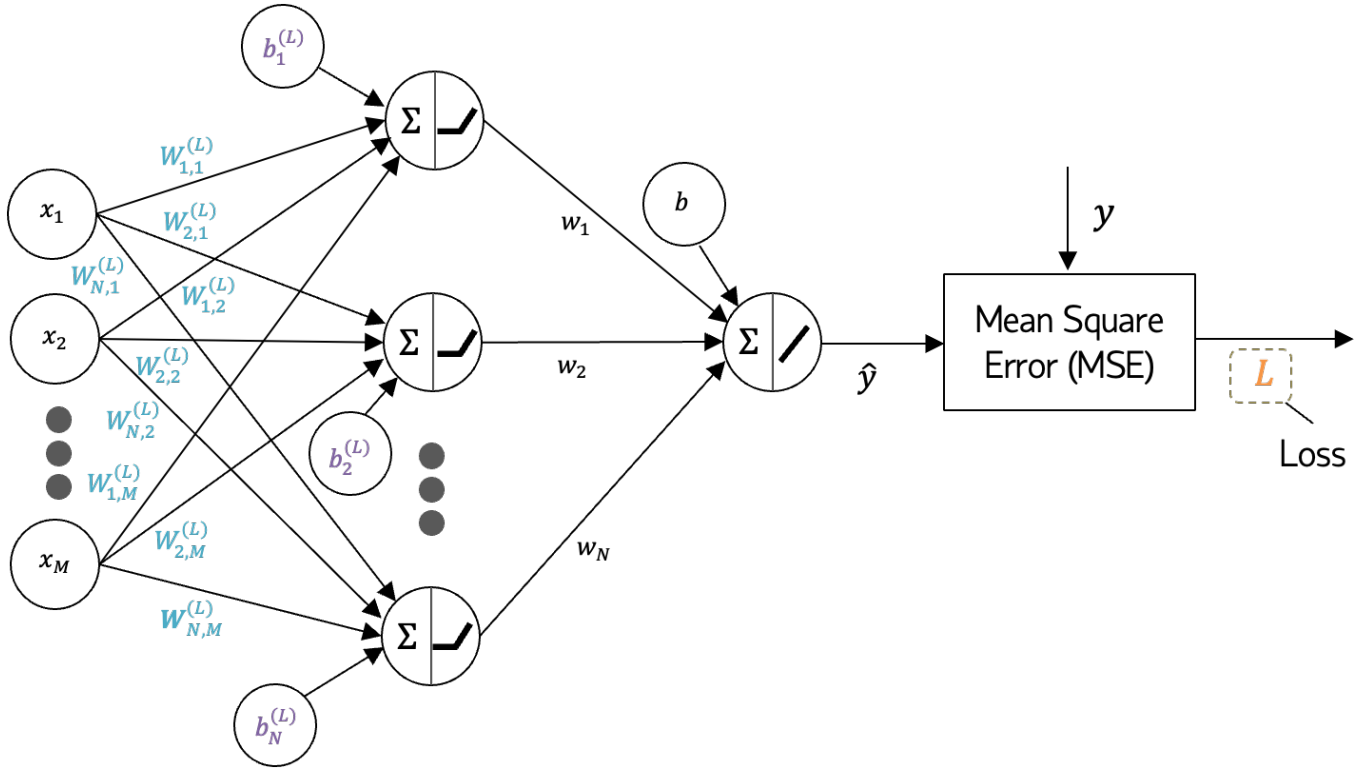
"How is $b_i^{(L)}$ affecting $\phi_i^{(L)}$?"

$$\frac{\partial L}{\partial b_i^{(L)}} = \frac{\partial L}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial \phi_i^{(L)}}\frac{\partial \phi_i^{(L)}}{\partial b_i^{(L)}}$$

"How is $\hat{y}$ affecting L?"

"How is $\phi_i^{(L)}$ affecting $\hat{y}$?"

Hidden Layer: Gradient Descent



Forward Path:

$$\Phi^{(L)}(\vec{x}) = \mathrm{ReLU}\left(\boldsymbol{W}^{(L)} \times \vec{x} + \vec{b}^{(L)}\right)$$

$$\hat{y} = \vec{w}^T \times \Phi^{(L)}(\vec{x}) + b$$

Mean Square Error:

$$L\left(\boldsymbol{W}, \vec{b}\right) = \frac{1}{B}\sum_{k=1}^{B}(y - \hat{y})^2$$

$$\frac{\partial L}{\partial W_{i,j}^{(L)}} = \frac{\partial L}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial \phi_i^{(L)}}\frac{\partial \phi_i^{(L)}}{\partial W_{i,j}^{(L)}} = (\hat{y} - y)w_i 1\left(\phi_i^{(L)} > 0\right)x_j$$

$$\frac{\partial L}{\partial b_i^{(L)}} = \frac{\partial L}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial \phi_i^{(L)}}\frac{\partial \phi_i^{(L)}}{b_i^{(L)}} = (\hat{y} - y)w_i 1\left(\phi_i^{(L)} > 0\right)$$

$\frac{\partial L}{\partial \hat{y}}$ and $\frac{\partial \hat{y}}{\partial \phi_i^{(L)}}$ were computed once, then reuse the values together to obtain $\frac{\partial L}{\partial W_{i,j}^{(L)}}$ and $\frac{\partial L}{\partial b_i^{(L)}}$ for all weights and biases."

Update: Step against the Slope

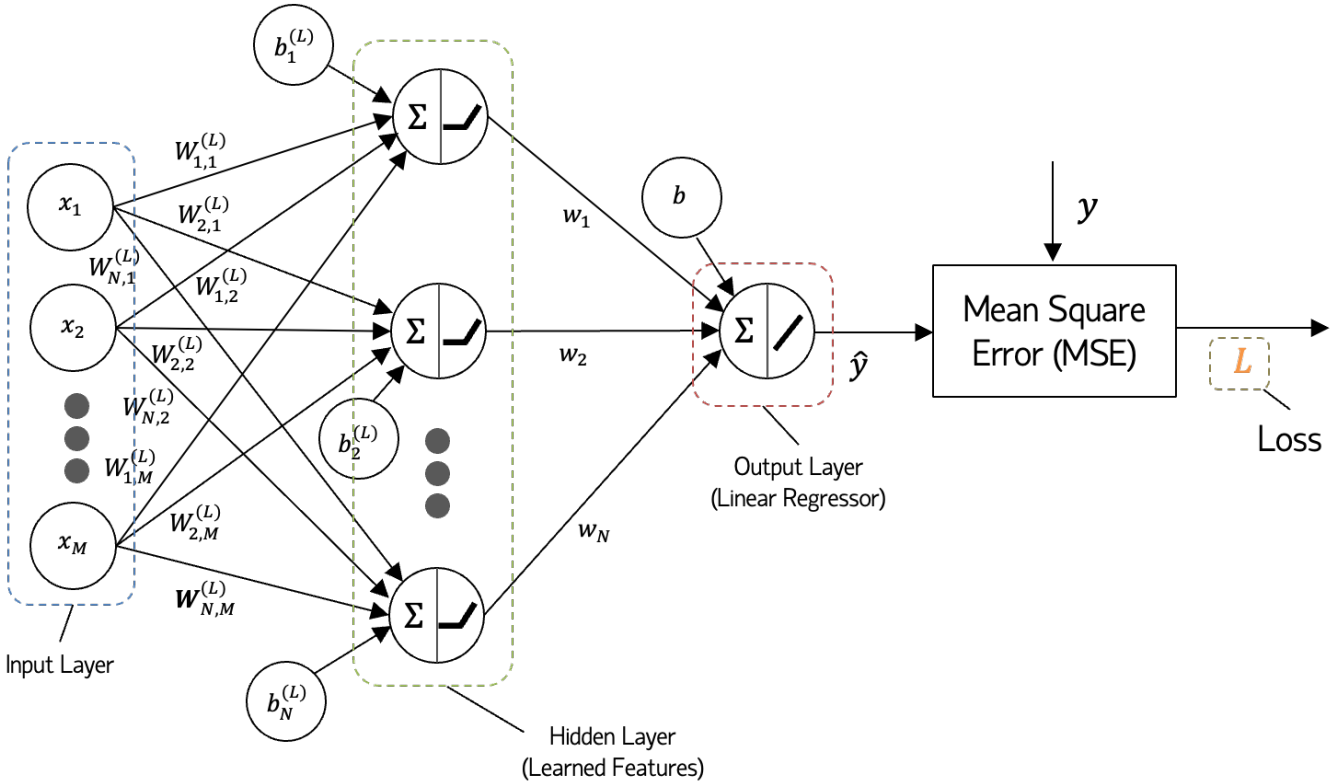
$$W_{i,j}^{(L)(t+1)} \leftarrow W_{i,j}^{(L)(t)} - \eta\frac{\partial L}{\partial W_{i,j}^{(L)(t)}}\left(W_{i,j}^{(L)(t)}\right)$$

$$\leftarrow W_{i,j}^{(L)(t)} - \eta \times \left(\frac{2}{B}\sum_{k=1}^{B}\left(\hat{y}^{(k)} - y^{(k)}\right)w_i 1\left(\phi_i^{(L)(k)} > 0\right)x_j^{(k)}\right)$$

$$b_i^{(L)(t+1)} \leftarrow b_i^{(L)(t)} - \eta\frac{\partial L}{\partial b}\left(b^{(t)}\right)$$

$$\leftarrow b^{(t)} - \eta \times \left(\frac{2}{B}\sum_{k=1}^{B}\left(\hat{y}^{(k)} - y^{(k)}\right)w_i 1\left(\phi_i^{(L)(k)} > 0\right)\right)$$

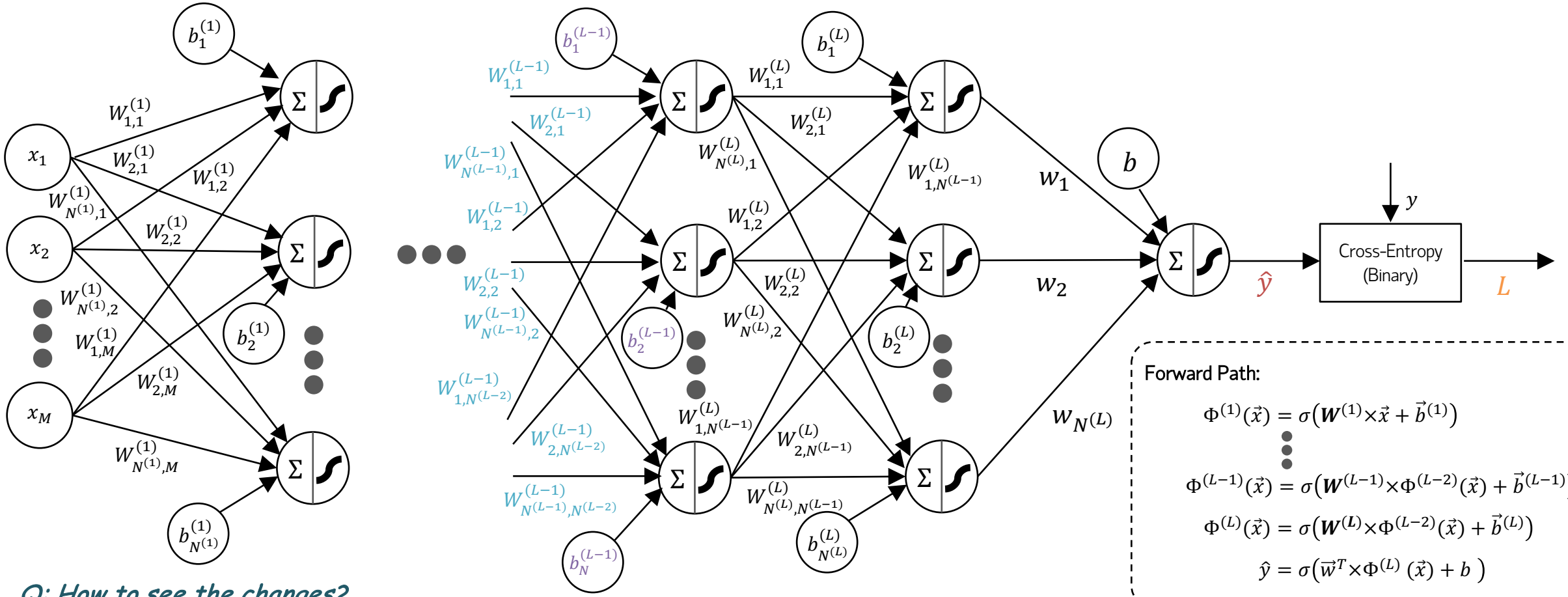
Tensorflow Code Snippet




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

num_inputs = M          # input dimension
num_hidden_units = N    # hidden units
num_outputs = 1         # hidden units


model = Sequential([
    Dense(num_hidden_units,
          activation="relu",
          input_shape=(num_inputs,)),

    Dense(num_outputs,
          activation="linear")
])

model.compile(optimizer="sgd",
              loss="mean_square_error",
              metrics=["mean_square_error"])
```

TensorFlow

Back Propagation: Deep Network

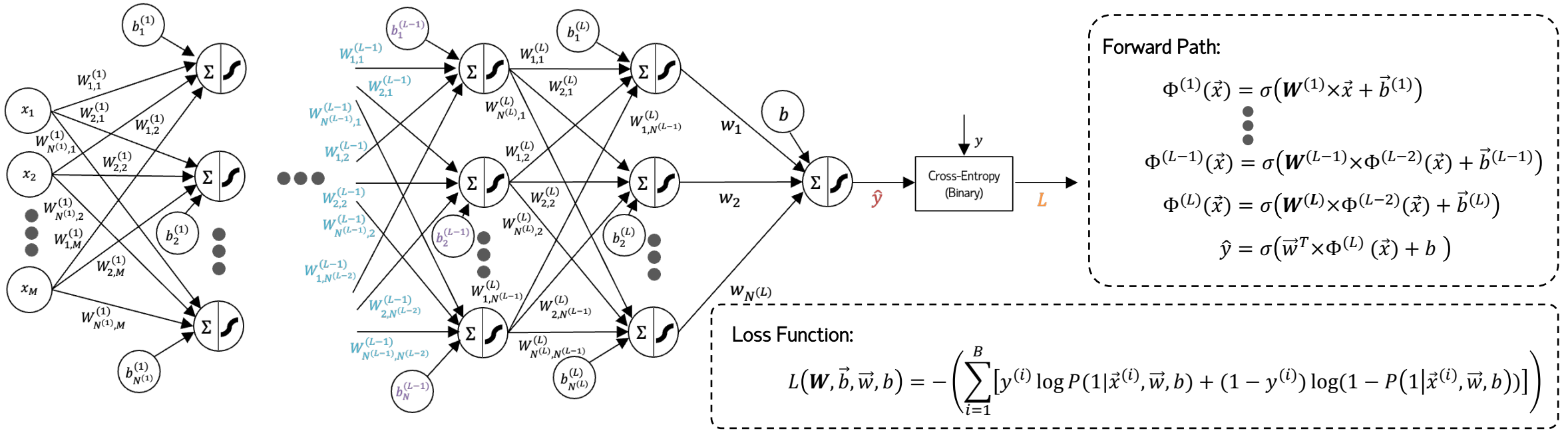


Q: How to see the changes?

A: Take the Slopes: (Chain-Rule: "Loss → Output → Features → (Low-Level) Features → Weight/Bias")

$$\frac{\partial L}{\partial W_{i,j}^{(L-1)}} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \phi_i^{(L)}} \frac{\partial \phi_i^{(L)}}{\partial \phi_i^{(L-1)}} \frac{\partial \phi_i^{(L-1)}}{\partial W_{i,j}^{(L-1)}} \quad \text{and} \quad \frac{\partial L}{\partial b_i^{(L-1)}} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \phi_i^{(L)}} \frac{\partial \phi_i^{(L)}}{\partial \phi_i^{(L-1)}} \frac{\partial \phi_i^{(L-1)}}{\partial b_i^{(L-1)}}.$$

Back Propagation: Algorithmic Nickname



Forward Pass: Each layer computes its output from the previous layer's numbers.

The calculation re-uses the gradients found layers earlier.

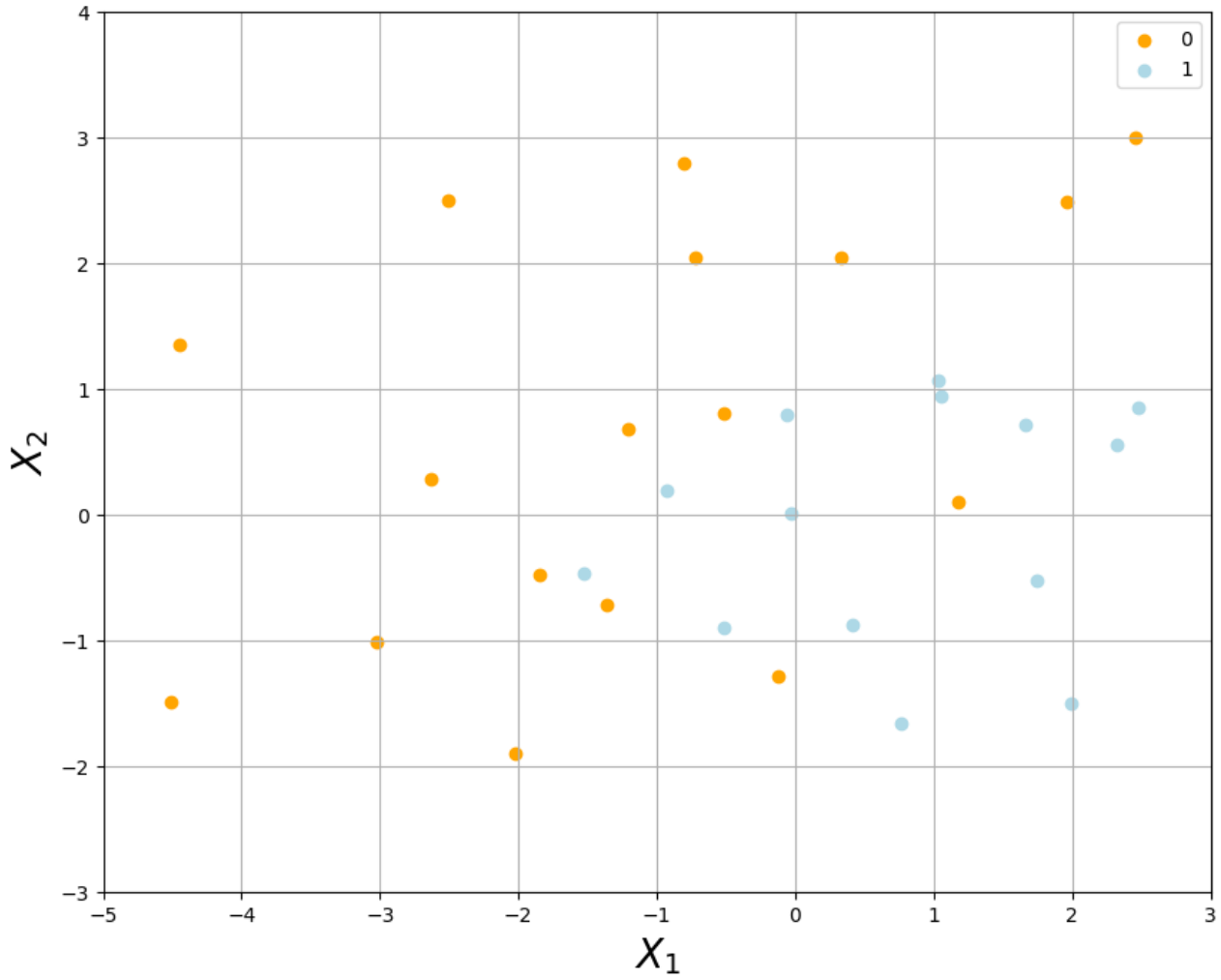
Backward Pass (Back-Prop): We now treat that loss as a function of every weight. Using the chain rule we calculate—layer by layer, in reverse order—the partial derivative of the loss with respect to each weight.

$$\frac{\partial L}{\partial W_{i,j}^{(L-1)}} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \phi_i^{(L)}} \frac{\partial \phi_i^{(L)}}{\partial \phi_i^{(L-1)}} \frac{\partial \phi_i^{(L-1)}}{\partial W_{i,j}^{(L-1)}} \quad \textit{and} \quad \frac{\partial L}{\partial b_i^{(L-1)}} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \phi_i^{(L)}} \frac{\partial \phi_i^{(L)}}{\partial \phi_i^{(L-1)}} \frac{\partial \phi_i^{(L-1)}}{\partial b_i^{(L-1)}}.$$

"Nothing physically travels backward."

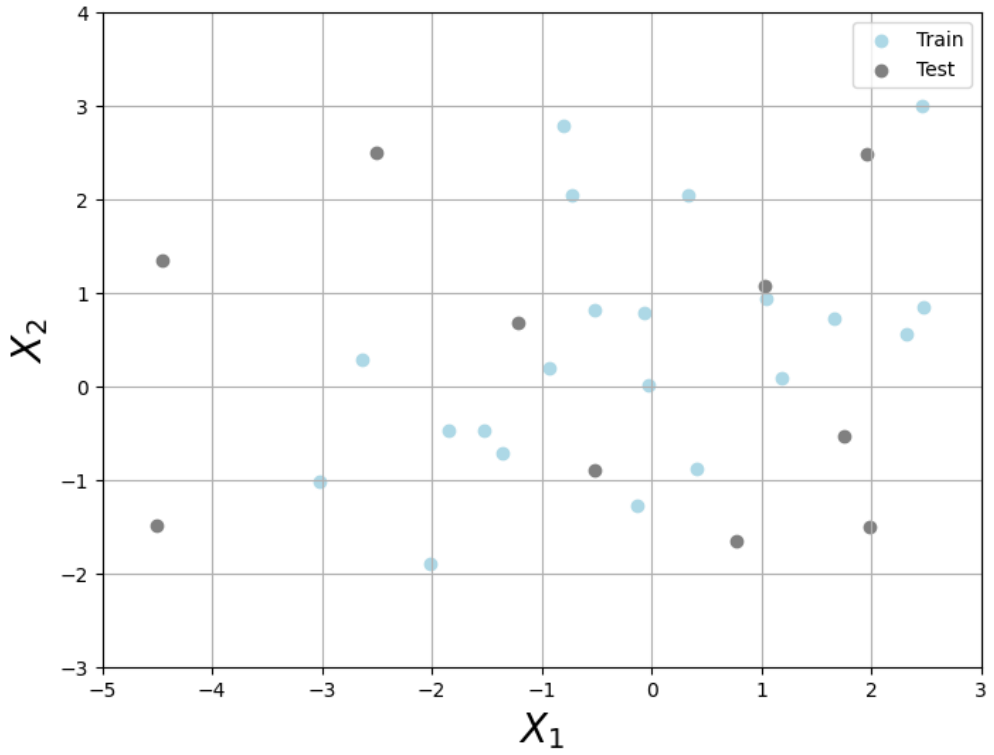
So "propagation" refers to how the gradient value is computed once at the output and then (re)used layer-by-layer all the way back to the earliest weights.

Example Dataset



Train, Validation and Test Datasets

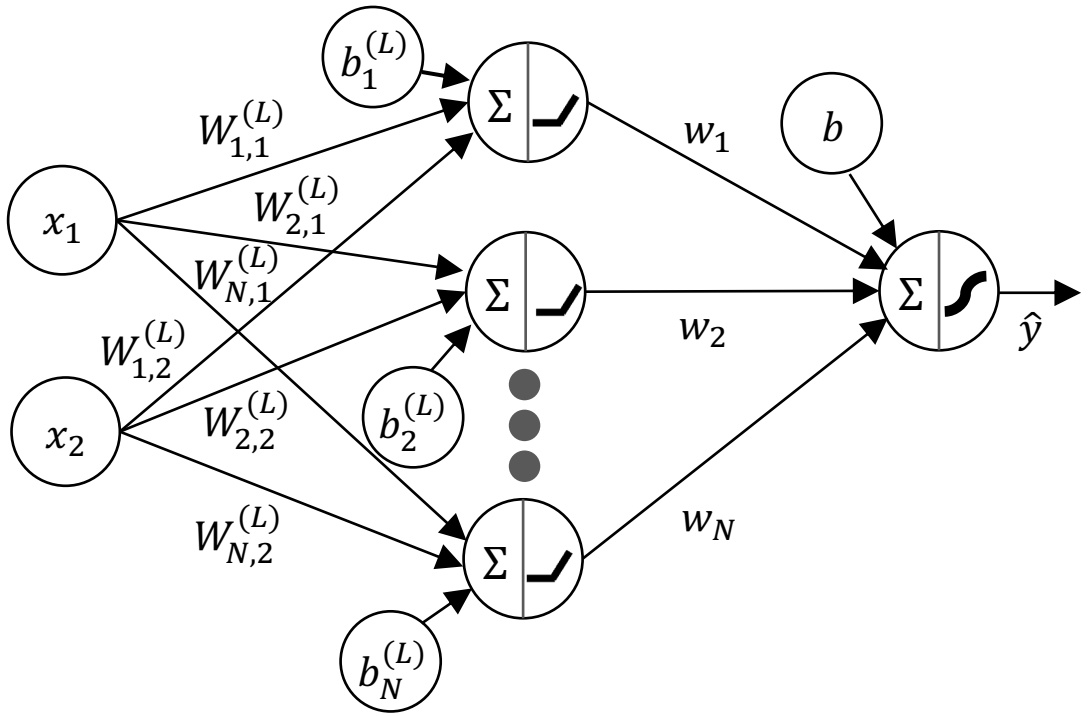
```
from sklearn.model_selection import train_test_split

# First, split the data into train (70%) and test (30%)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42, stratify=y)
```



- Train:Test is typicaly either 0.8:0.2 or 0.7:0.3.
- **Test** dataset is a proxy of unseen data, and it will only be used in the final evaluation.
- **Train** dataset is further divided into k folds (e.g., 5 or 10).For each fold, the model is trained on k-1 folds and validated on the remaining fold.
- We use **K-Fold Cross-Validation** to fine-tune or optimize the ML model. Here, we find an optimal NN architecture based on the average performance across folds.

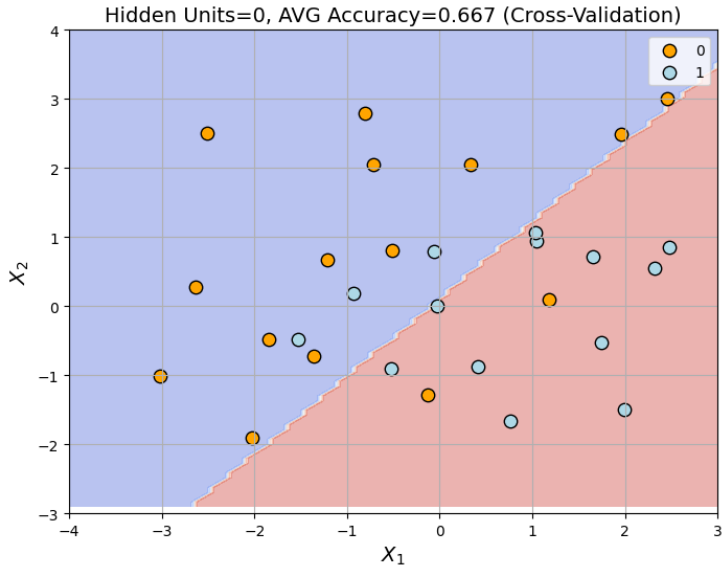
Tensorflow Code Snippet

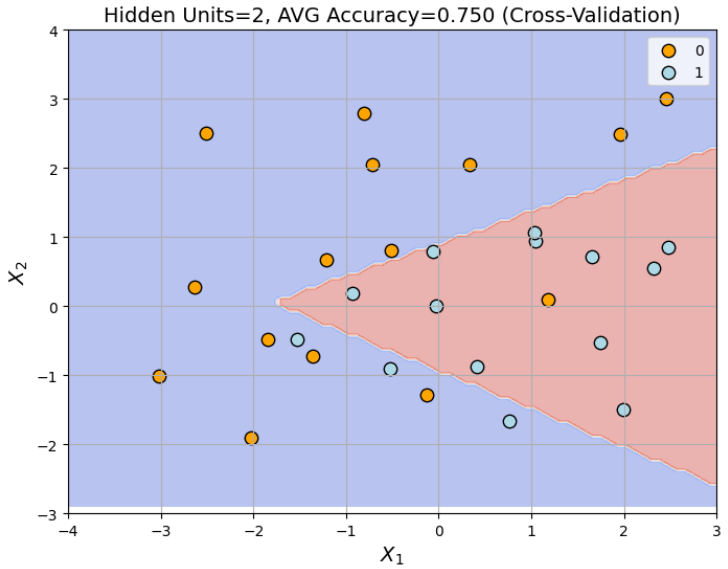
```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
import numpy as np
from sklearn.model_selection import KFold

num_inputs = 2          # input dimension
num_hidden_units = N    # hidden units
num_outputs = 1         # output dimension

def build_model():
    model = Sequential([
                Dense(num_hidden_units,
                    activation=”relu",
                     input_shape=(num_inputs,)),

                Dense(num_outputs,
                    activation="sigmoid”)
            ])

    model.compile(optimizer=”sgd",
                  loss="binary_crossentropy",
                  metrics=["accuracy"])

    return model

kfold = KFold(n_splits=5, shuffle=True, random_state=42)

val_scores = []
for train_idx, val_idx in kfold.split(X):
    X_train, X_val = X[train_idx], X[val_idx]
    y_train, y_val = y[train_idx], y[val_idx]

 … cont …
```

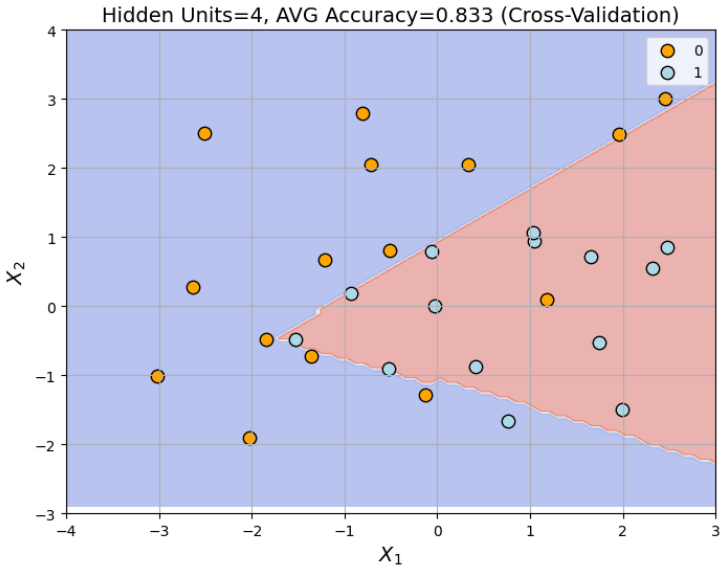


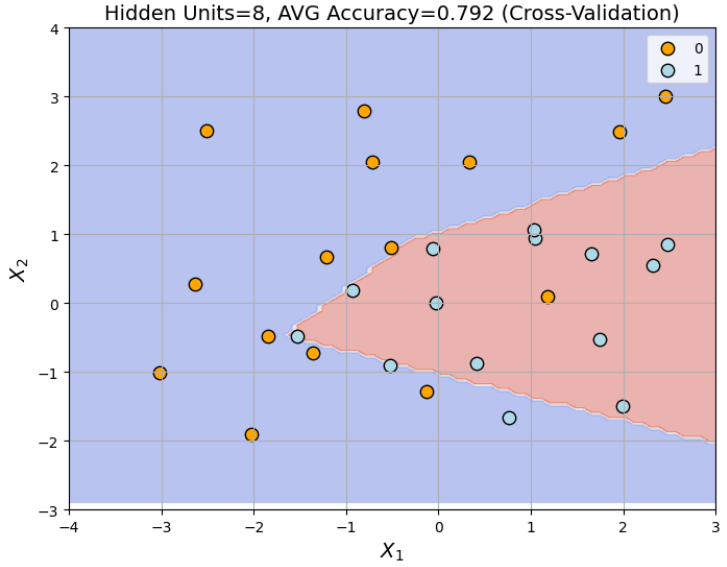

```
    model = build_model()      # fresh weights each fold
    model.fit(X_train, y_train,
              epochs=20, batch_size=128,
              verbose=0)

    loss, acc = model.evaluate(X_val, y_val, verbose=0)
    val_scores.append(acc)

print("Fold accuracies:", val_scores)
print("Mean ± std    :", np.mean(val_scores), "±", np.std(val_scores))
```

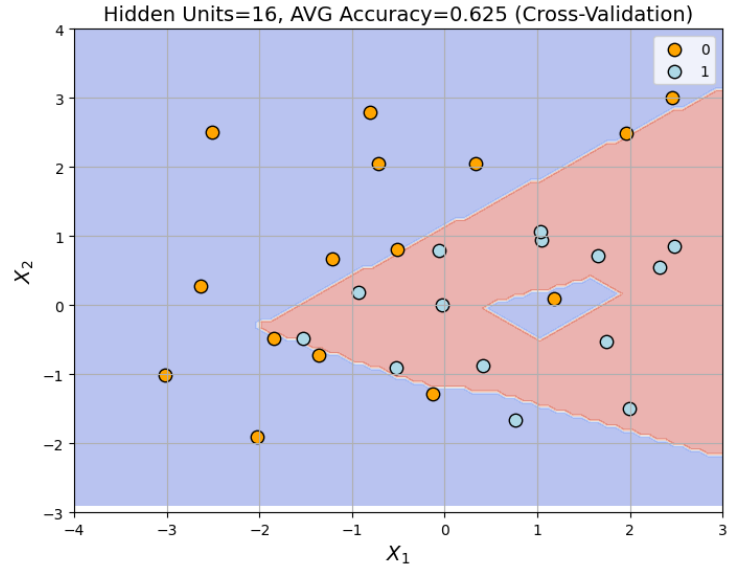
Model Complexity

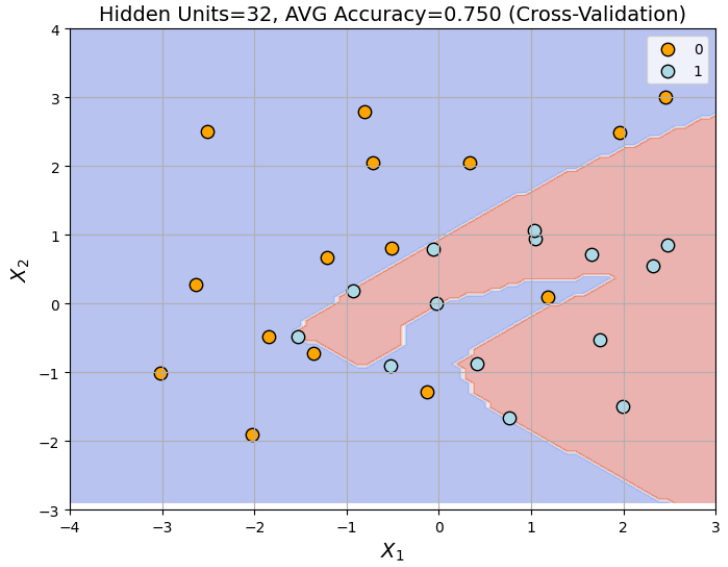




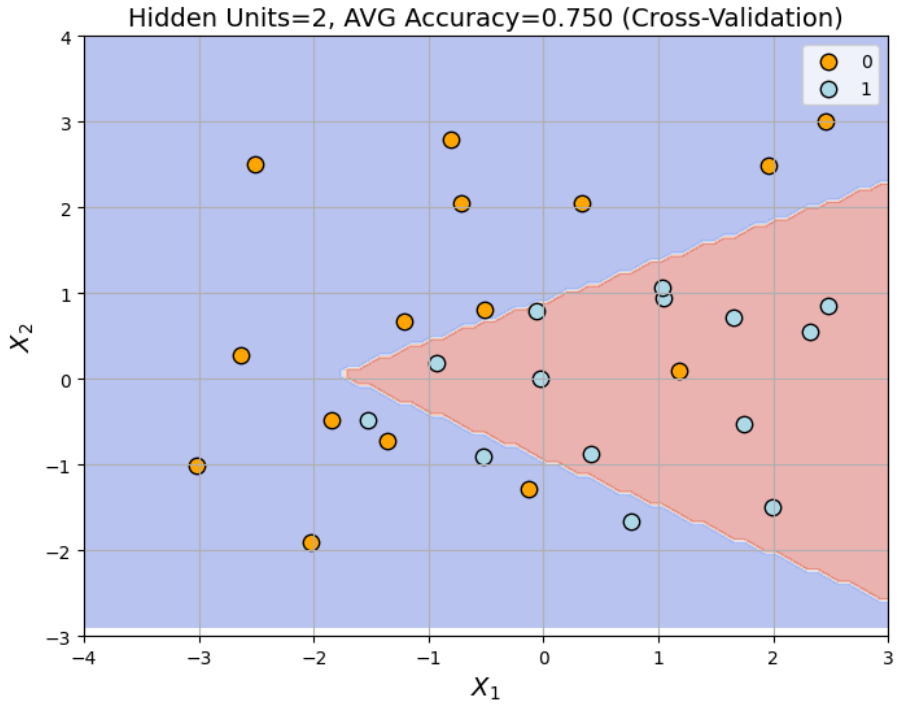






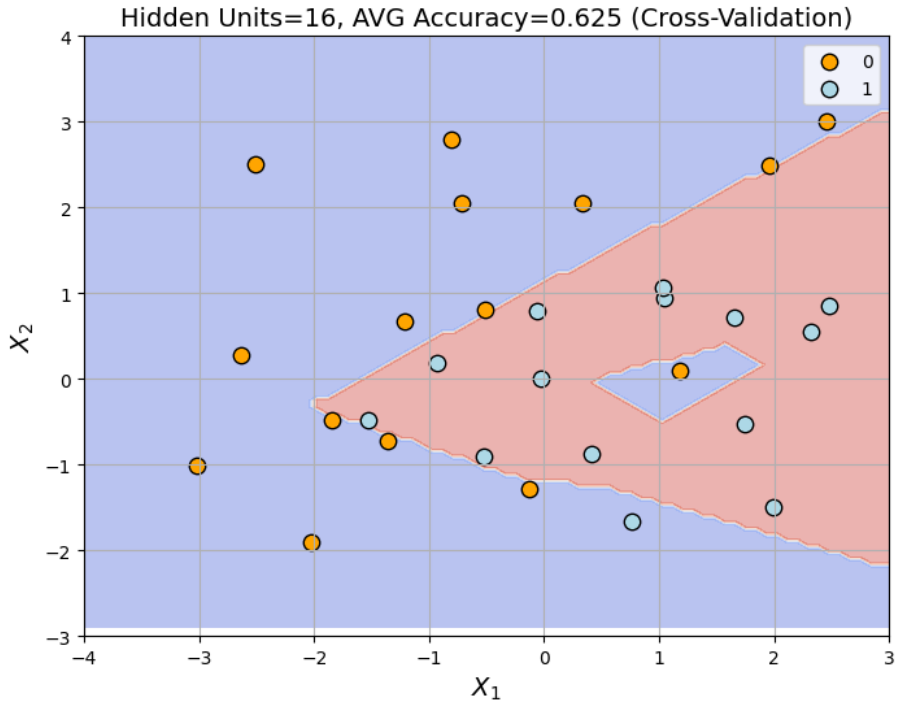


When More Units Hurt, Fewer Units Help

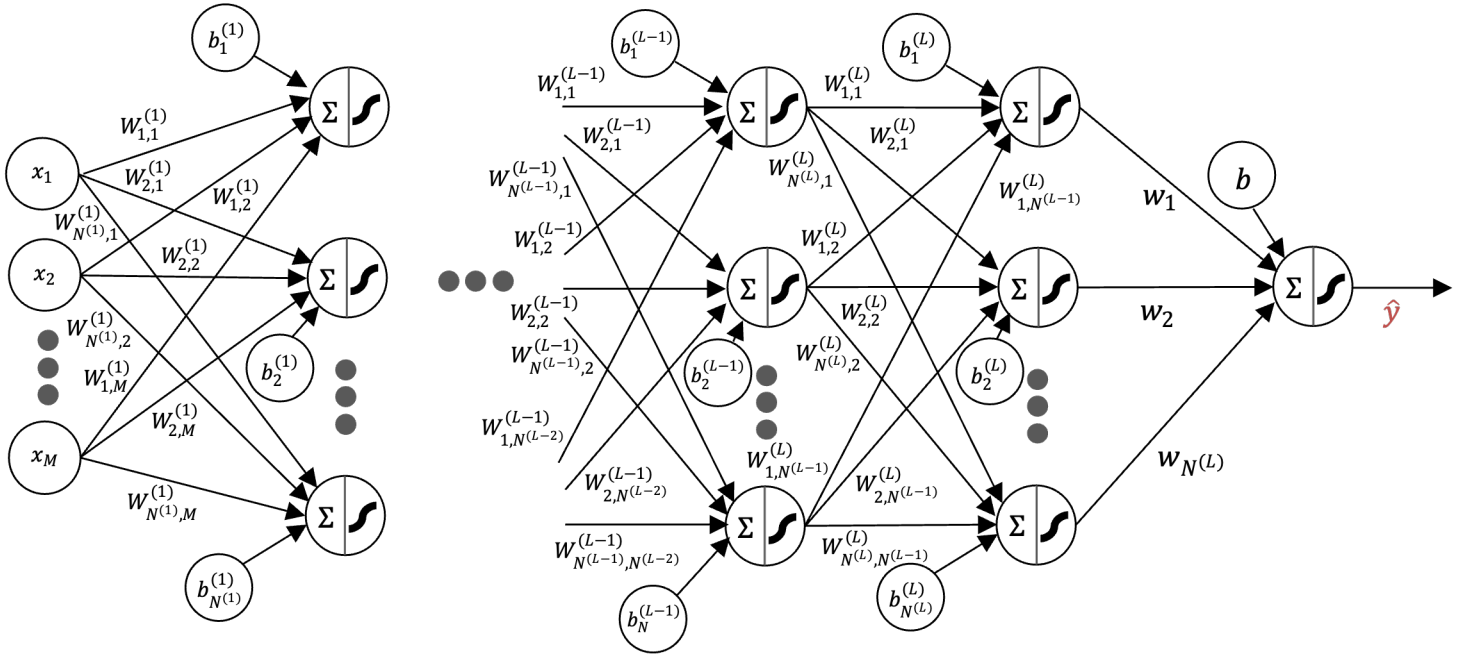


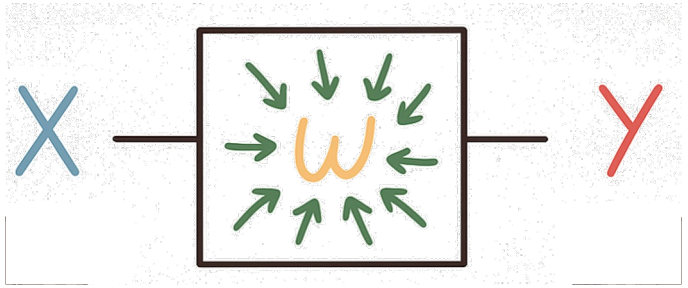
- Hidden Units: 2
- $\mathrm{MIN}(\boldsymbol{W}) = -17.71, \mathrm{MAX}(\boldsymbol{W}) \leq 19.54$
- Simpler Decision Boundary ← Just Enough Capacity
- High Validation Accuracy (0.750)

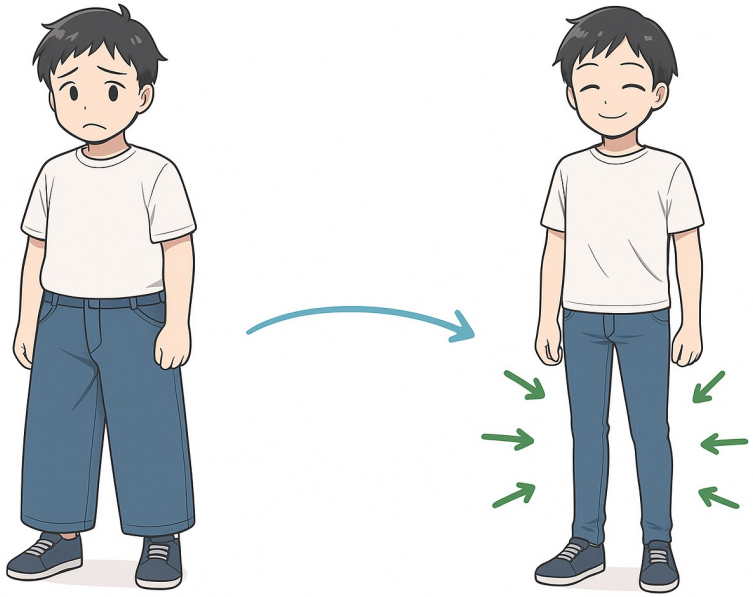
- Hidden Units: 16
- Too Many Trainable Parameters
- $\mathrm{MIN}(\boldsymbol{W}) = -59.63, \mathrm{MAX}(\boldsymbol{W}) \leq 50.71$
- Memorising Training Data → Complex Decision Boundary
- Low Validation Accuracy (0.625)



Regularisation







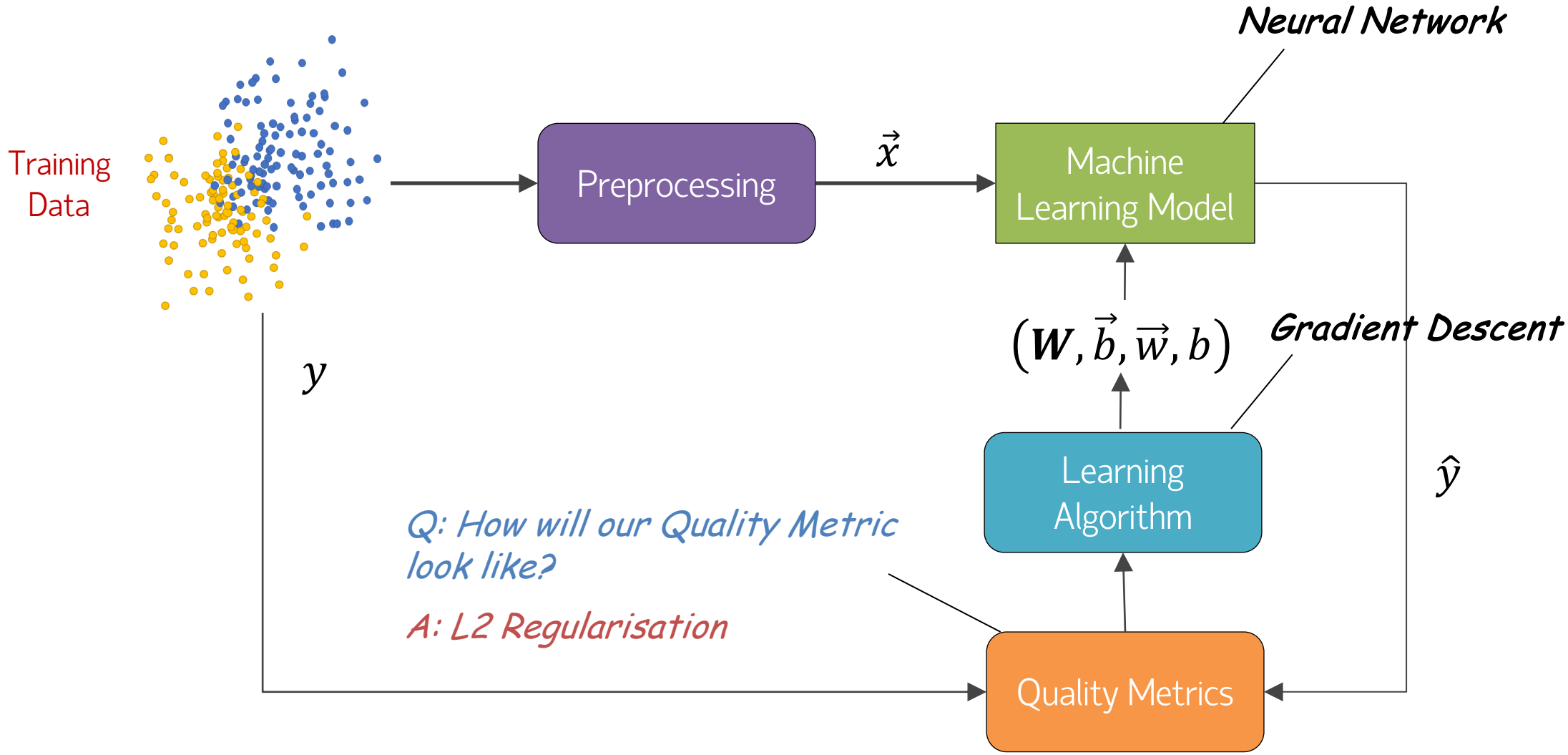
Start Big: *Use a large (Deep & Wide) network so no potentially useful pattern is missed..*

Risk: *A big network can* ***over-fit****, memorising training noise.*

Fix: *Add regularisation (weight-decay, dropout, etc.) to* ***shrink the weights*** *toward zero.*

Result: *Smaller weights → simpler model →* ***less over-fitting*** *→ better generalisation.*

Workflow: Neural Network



Desired Total Cost

Want to Balance

Total Cost = Measure of Fit + Measure of Magnitude of Weights

$$L(\boldsymbol{W}, \vec{b}, \vec{w}, b)$$

E.g. (Binary) Cross-Entropy, Mean Square Error (MSE)

$$\|\boldsymbol{W}\|_2^2 + \|\vec{w}\|_2^2 = (W_{1,1}^{(1)})^2 + \ldots + (W_{N^{(L)},N^{(L-1)}}^{(L)})^2 + (w_1)^2 + \cdots + \left(w_{N^{(L)}}\right)^2$$

Resulting Objective

Minimising Total Cost Function:

$$\min_{\boldsymbol{W},\vec{b},\vec{w},b} L\left(\boldsymbol{W},\vec{b},\vec{w},b\right) + \lambda(\|\boldsymbol{W}\|_2^2 + \|\vec{w}\|_2^2)$$

Penalty Parameter, i.e. Balance of Fit and Magnitude

L2 Regularisation

US Airline Sentiment Dataset

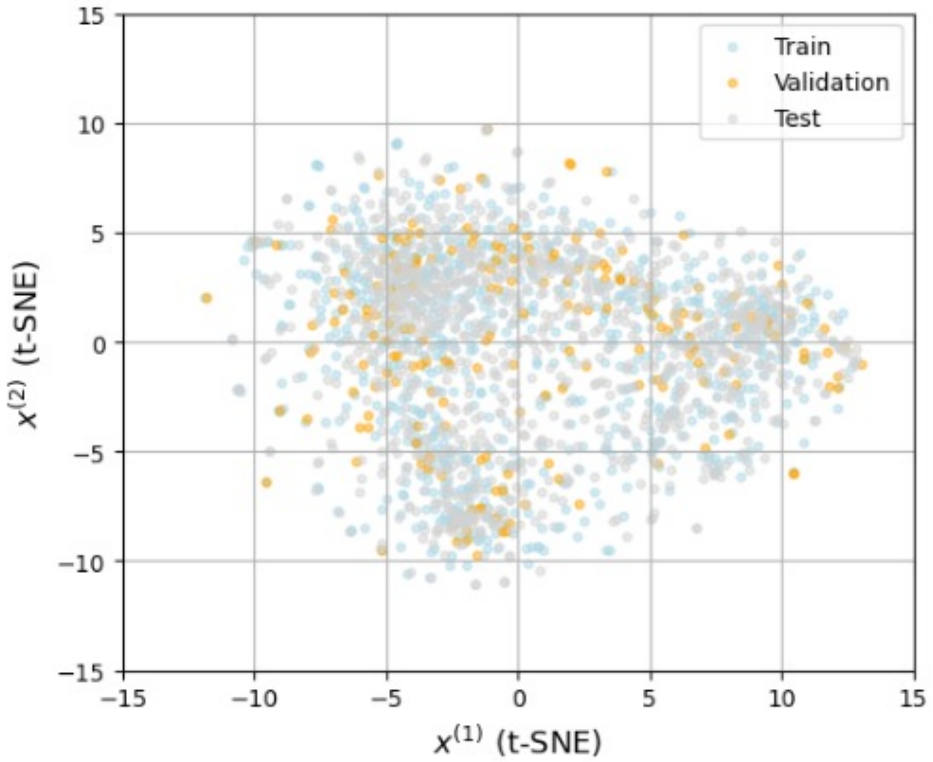
Key Facts	Details
Size	≈ 14.6 k English-language tweets (collected 2013-2015).
Target Label	airline_sentiment ∈ { negative (≈63 %), neutral (≈21 %), positive (≈16 %) }.
Airlines Covered	American, United, Southwest, Delta, US Airways, Virgin America.
Main Text Field	text – the raw tweet.
Auxiliary Fields	Tweet ID, time stamp, user handle, airline name, sentiment confidence, tweet latitude/longitude, etc

```
@united delayed again. Sat on the tarmac for an hour,
missed my connection, customer service no help.
```

```
Huge thanks to @SouthwestAir for getting me home
early & with a smile. You guys always deliver!
```

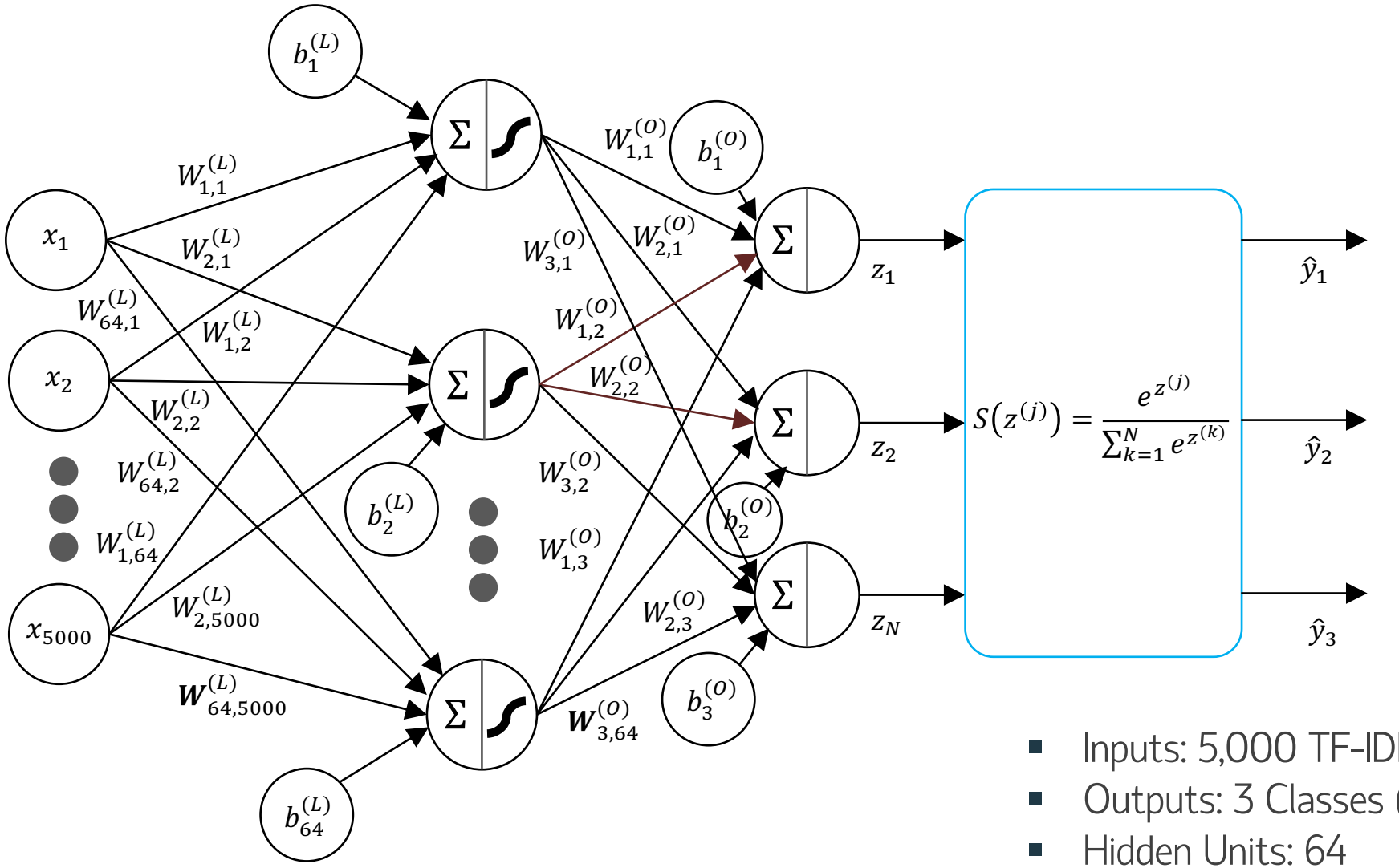
Train, Validation and Test Datasets

```
from sklearn.model_selection import train_test_split

# First, split the data into train (60%) and temp (40%)
X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.4, random_state=42, stratify=y)

# Then split the temp data into validation (50% of 40% ⇒ 20% of total) and test (remaining 50% of 40% ⇒ 20% of total)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42, stratify=y_temp)
```



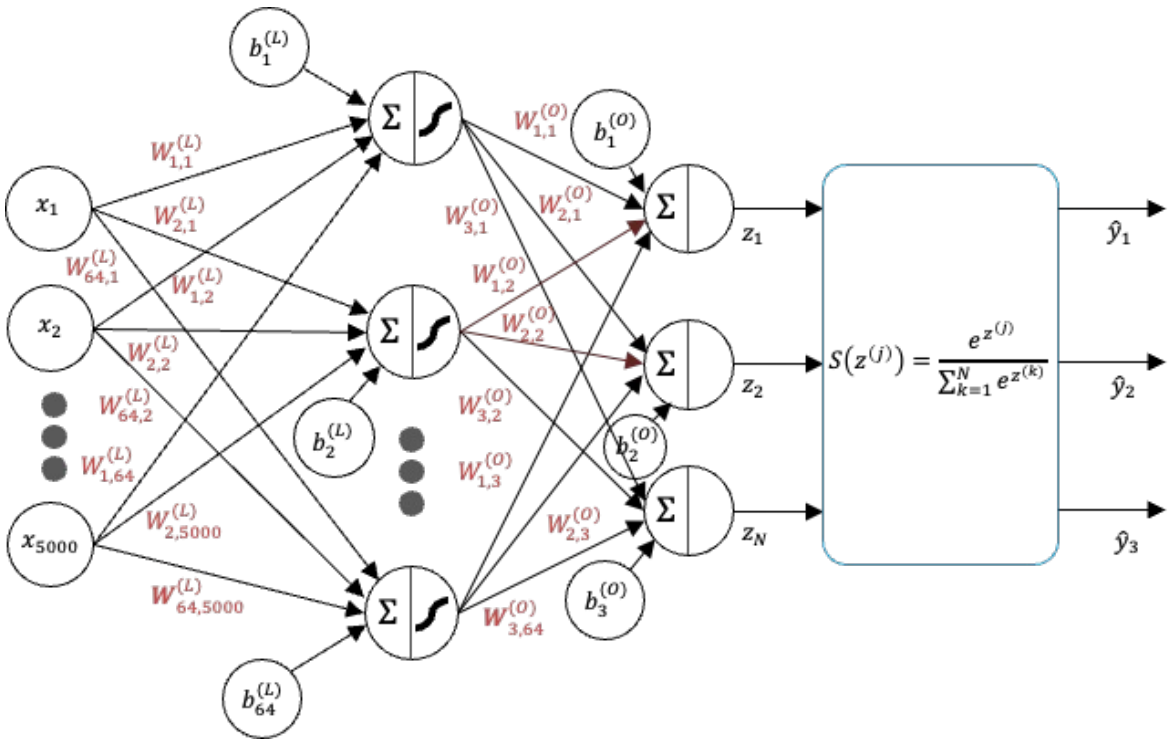
- In DL context, we prefer a large network so useful patterns will not be missed, hence computationally expensive.
- With a large dataset, we can simply keep out a hold-out validation set instead of running cross-validation.
- **Using Validation** datest, we find λ (L2 regularization) which will result in the best validation performance. We can also evaluate different neural network architectures.

Multi-Class Classifier: Sentiment Analysis



- Inputs: 5,000 TF-IDF Tokens
- Outputs: 3 Classes (One-Hot Encoded)
- Hidden Units: 64

Tensorflow Code Snippet



L2 keeps the weights small.

```
@united delayed again. Sat on the tarmac for an hour,
missed my connection, customer service no help.
```

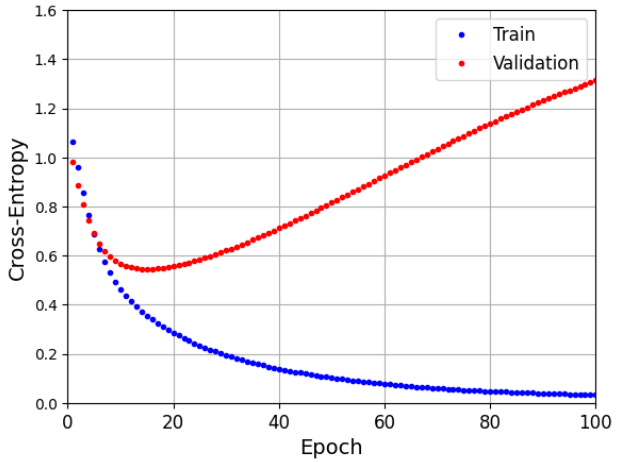
```
Huge thanks to @SouthwestAir for getting me home
early & with a smile. You guys always deliver!
```

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.regularizers import l2
import tensorflow_addons as tfa

num_inputs = 5000        # input dimension
num_hidden_units = 64    # hidden units
num_outputs = 3          # output dimension
penalty = 1e-3           # λ

f1 = tfa.metrics.F1Score(num_classes=num_outputs, average="macro")

model = Sequential([
    Dense(num_hidden_units,
          activation="sigmoid",
          kernel_regularizer=l2(penaly),      # ← regularise weights
          input_shape=(num_inputs,)),

    Dense(num_outputs,
          activation="softmax",
          kernel_regularizer=l2(l2_strength)) # ← regularise weights
])

model.compile(optimizer=”sgd",
              loss=”categorical_crossentropy",
              metrics=[f1])

history = model.fit(X_train, y_train,
                    batch_size=512,
                    epochs=100,
                    validation_data=(X_val, y_val))
```

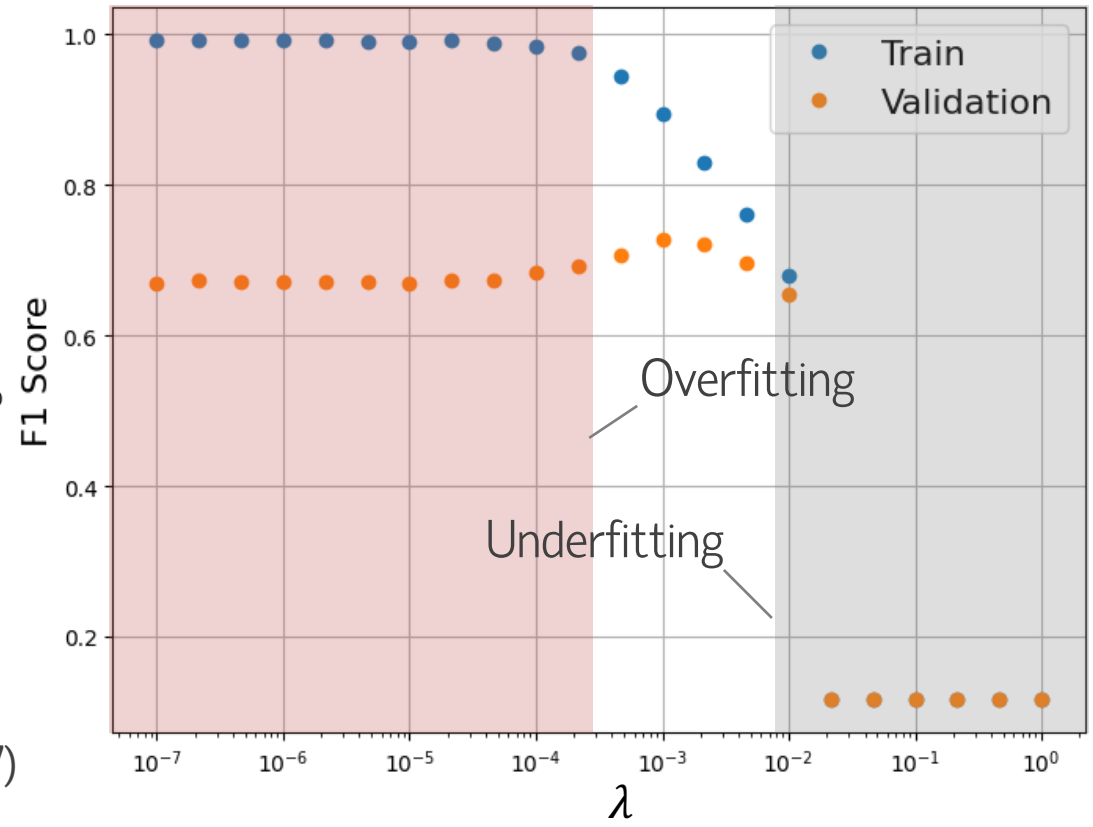
TensorFlow

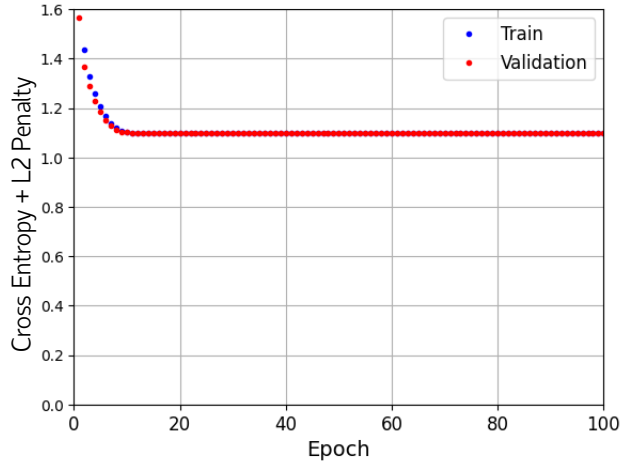
Overfitting vs Underfitting

- Small λ (weak weight-decay) – weights can still grow large, so the network ends up memorising the training set. F1 (Training) stays high while vF1 (Validation) stays lower, an indication of over-fitting.
- Large λ (strong weight-decay) – the penalty pushes weights close to zero, leaving the model too simple to capture real patterns. Both F1s on training and validation datasets remain low, showing under-fitting.



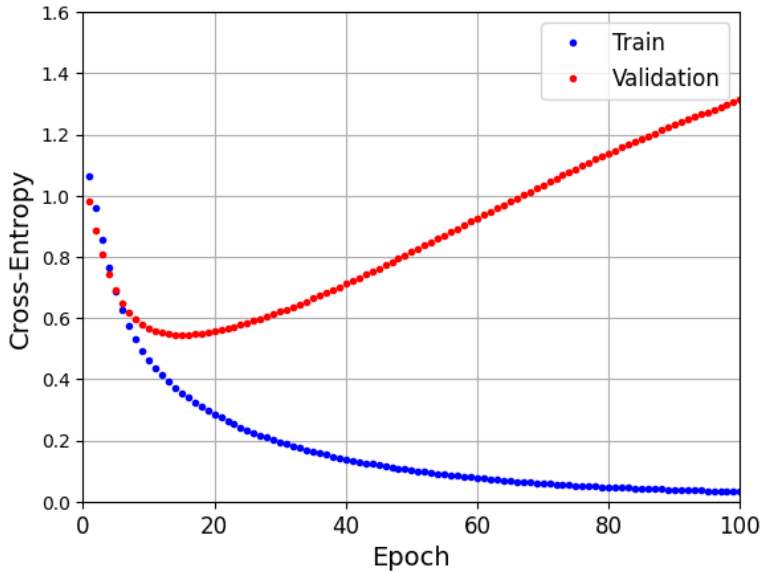
- No Weight Constraint, $\|W\|_2 = 44.36$
- Getting Better at Memorising Training Data
- High Train F1 Score (0.99)
- Low Validation F1 Score (0.67)



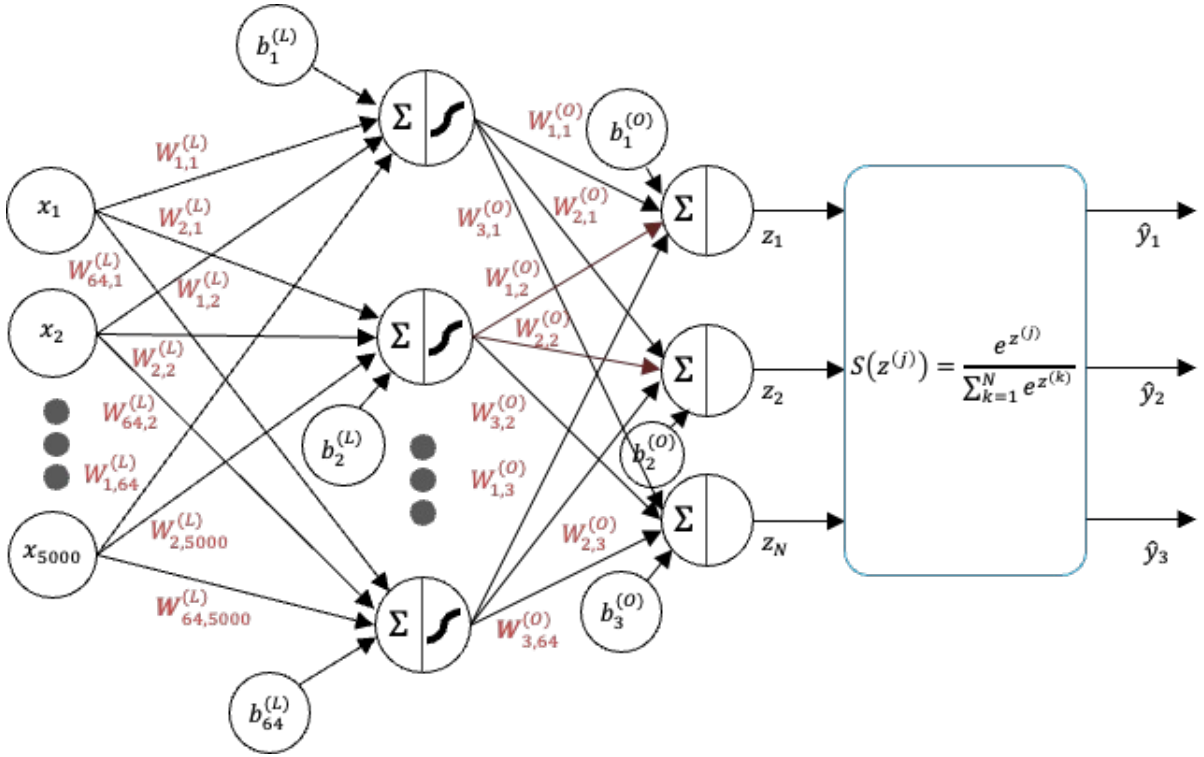


- High Weight Constraint, $\|W\|_2 = 0.03$
- Unable to Learn Intricated Relationships
- Low F1 Scores on both Train (0.12) and Validation (0.12) Data

Tensorflow Code Snippet



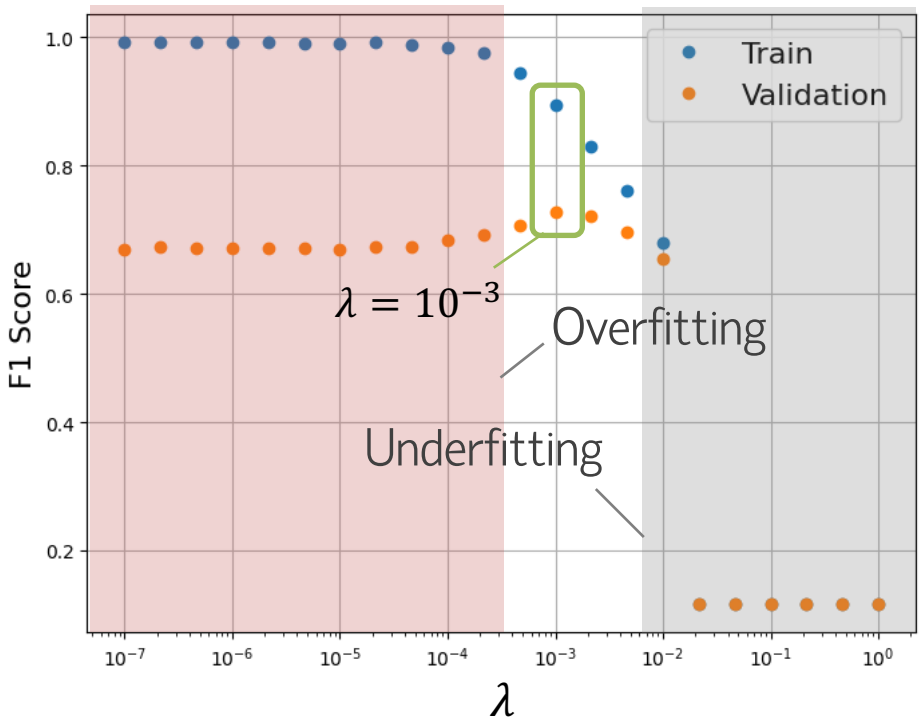
- No Weight Constraint, $\|W\|_2 = 44.36$
- Getting Better at Memorising Training Data
- High Train F1 Score (0.99)
- Low Validation F1 Score (0.67)

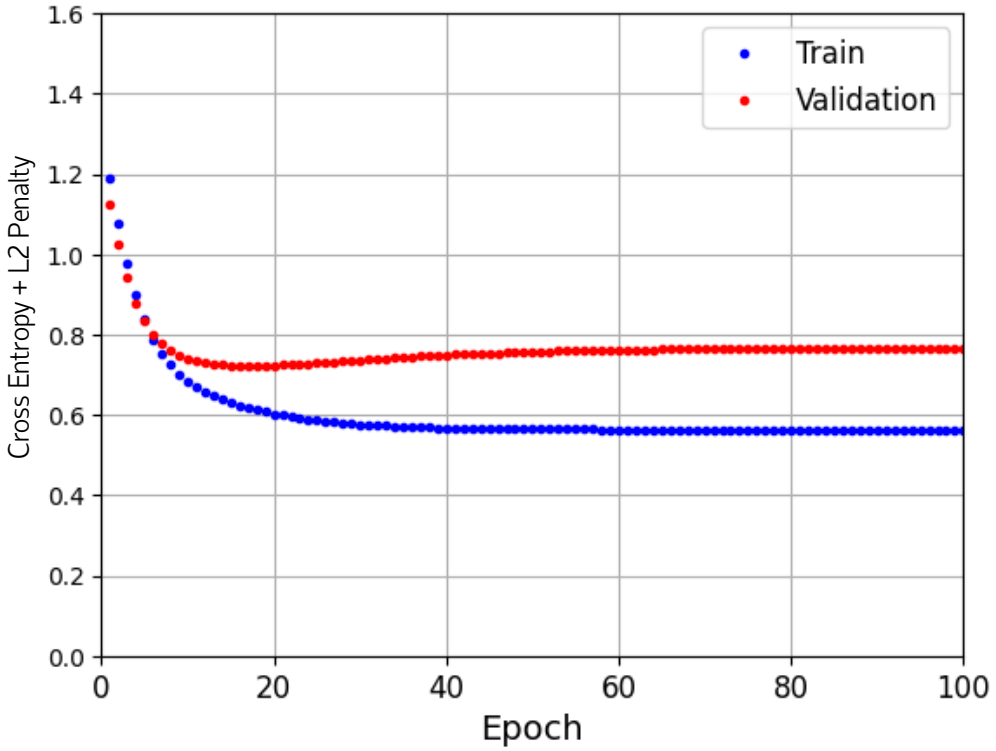



```
flat = np.concatenate([w.numpy().ravel()
                       for w in model.trainable_weights
                       if "kernel" in w.name])
w_stats = dict(min=flat.min(),
               max=flat.max(),
               l2 =np.linalg.norm(flat))
```

TensorFlow

Optimal Model





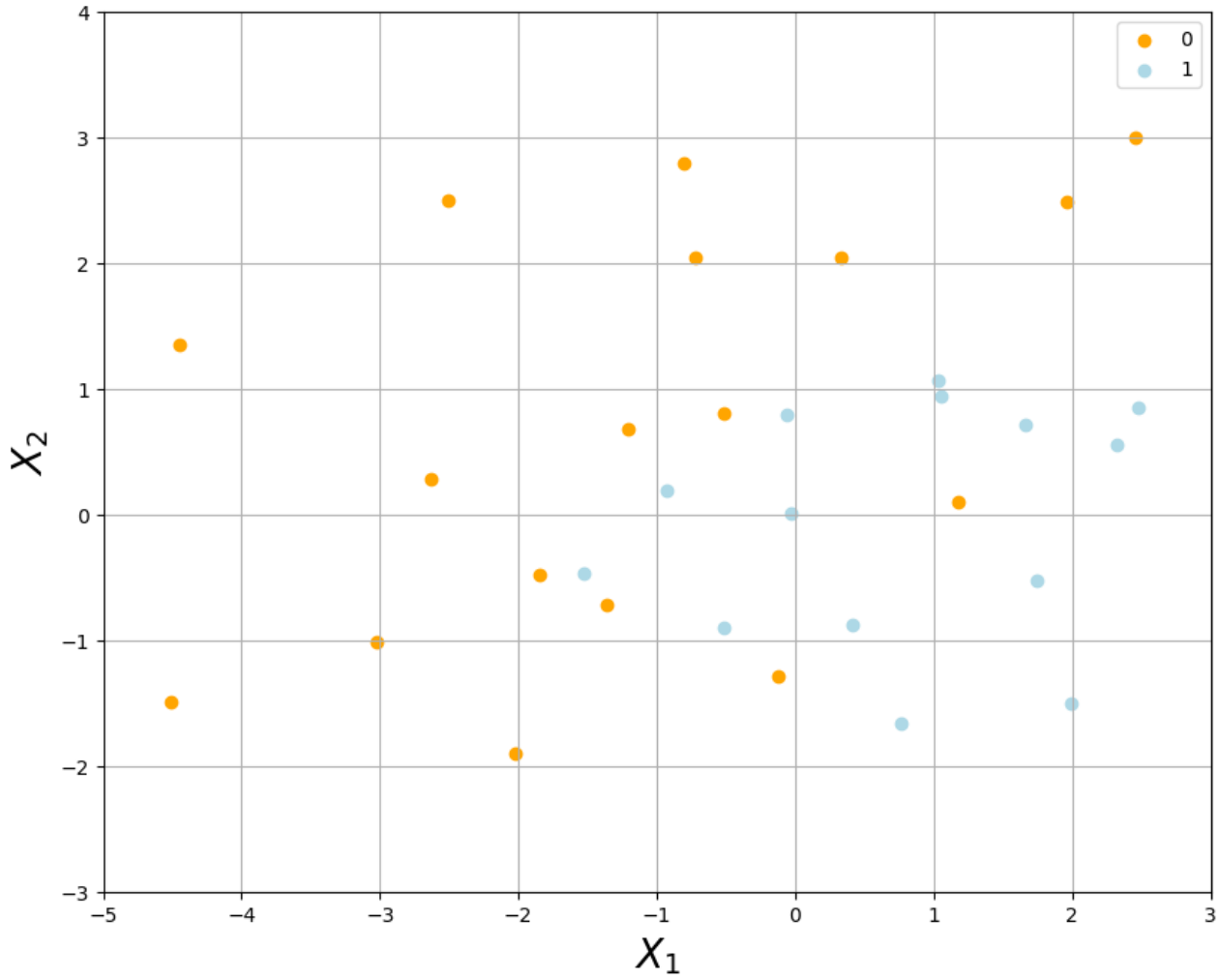
- Moderately Constrained on Weights, $\|W\|_2 = 14.78$
- Highest F1 Score (0.73) on Validation Data
- Train F1 Score (0.90)

Observation	Interpretation
Training loss keeps falling and levels out around **0.55**.	The model is still getting better at memorising the training data.
Validation loss bottoms out near **epoch ≈ 12–15** (~0.72) **then climbs** and plateaus around **0.77–0.78**.	After that point the network is starting to fit patterns that do **not** help on unseen data ⇒ classic sign of *over-fitting*.
The gap between the two curves stabilises (~0.20).	L2 regularisation is limiting runaway divergence, but it hasn't eliminated it.

Where to Use L2 Regularisation (and Why)

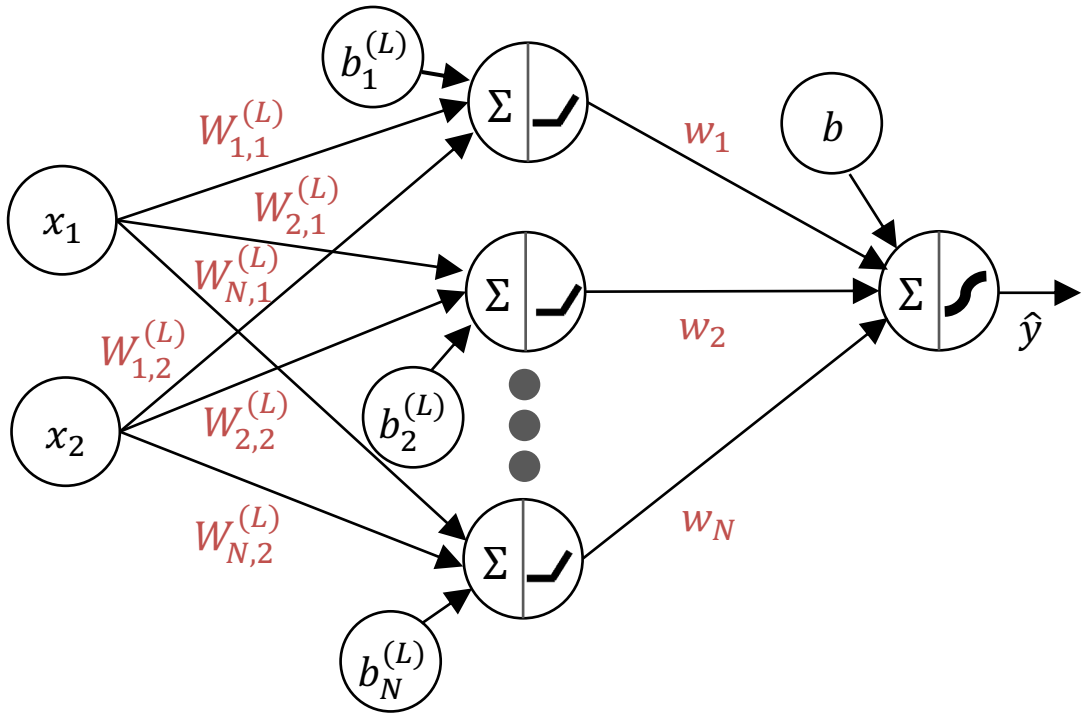
Layer / Parameter Set	Typical Practice	Why It Helps (or Why We Skip)
All weight matrices (Dense, Conv, RNN)	**Yes – default.** Add kernel_regularizer=l2(**λ**) (Keras) or set weight_decay (PyTorch) for every learnable weight tensor.	Penalises large weights ⇒ smoother functions, less over-fit, better numerical stability.
Bias vectors	*Often skipped.* (bias_regularizer=None)	Biases are few; shrinking them rarely changes capacity but costs extra computation.
Embedding tables / projection layers	**Yes, but with a smaller λ.**	Keeps vector norms bounded without erasing semantic information; small decay prevents collapse to zero.
Batch-Norm scale & shift (γ, β)	*Usually off.* (l2=0 on BN params)	BN already normalises activations; extra decay can fight the learned scale and slow convergence.
Recurrent kernels (LSTM/GRU)	**Yes.** Use the same **λ** as other weights.	Controls exploding weights in long-sequence training; complements gradient clipping.
Convolution kernels	**Yes.** (Same **λ** as dense)	Prevents a few filters from getting huge and dominating the feature map.
Very deep nets with Dropout	**Still yes.** Decay and dropout act on different objects (weights vs activations) and usually stack well.	
Models with heavy Batch-Norm & data-augmentation	**Consider reducing λ.** BN + strong aug already give robustness; large L2 may under-fit.	
Output layer weights	**Yes, harmless.** Keeps the final classifier head from growing giant logits and over-confident probabilities.	

Example Dataset: Revisit



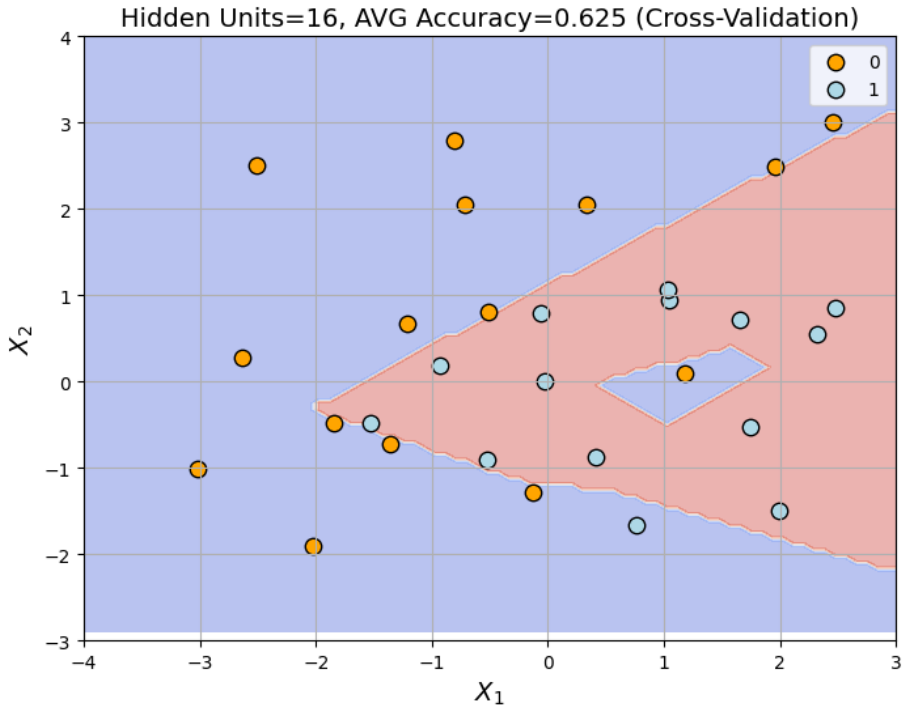
Tensorflow Code Snippet

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
import numpy as np
from sklearn.model_selection import KFold

num_inputs = 2          # input dimension
num_hidden_units = N    # hidden units
num_outputs = 1         # output dimension
penalty = 1e-4           # λ

def build_model():
    model = Sequential([
                Dense(num_hidden_units,
                      activation=”relu",
                      kernel_regularizer=l2(penaly),
                     input_shape=(num_inputs,)),
                Dense(num_outputs,
                      activation="sigmoid”),
                      kernel_regularizer=l2(penaly)
            ])
    model.compile(optimizer=”sgd",
                  loss="binary_crossentropy",
                  metrics=["accuracy"])
    return model

kfold = KFold(n_splits=5, shuffle=True, random_state=42)

val_scores = []
for train_idx, val_idx in kfold.split(X):
    X_train, X_val = X[train_idx], X[val_idx]
    y_train, y_val = y[train_idx], y[val_idx]

 … cont …
```




```
    model = build_model()      # fresh weights each fold
    model.fit(X_train, y_train,
              epochs=20, batch_size=128,
              verbose=0)

    loss, acc = model.evaluate(X_val, y_val, verbose=0)
    val_scores.append(acc)

print("Fold accuracies:", val_scores)
print("Mean ± std    :", np.mean(val_scores), "±", np.std(val_scores))
```

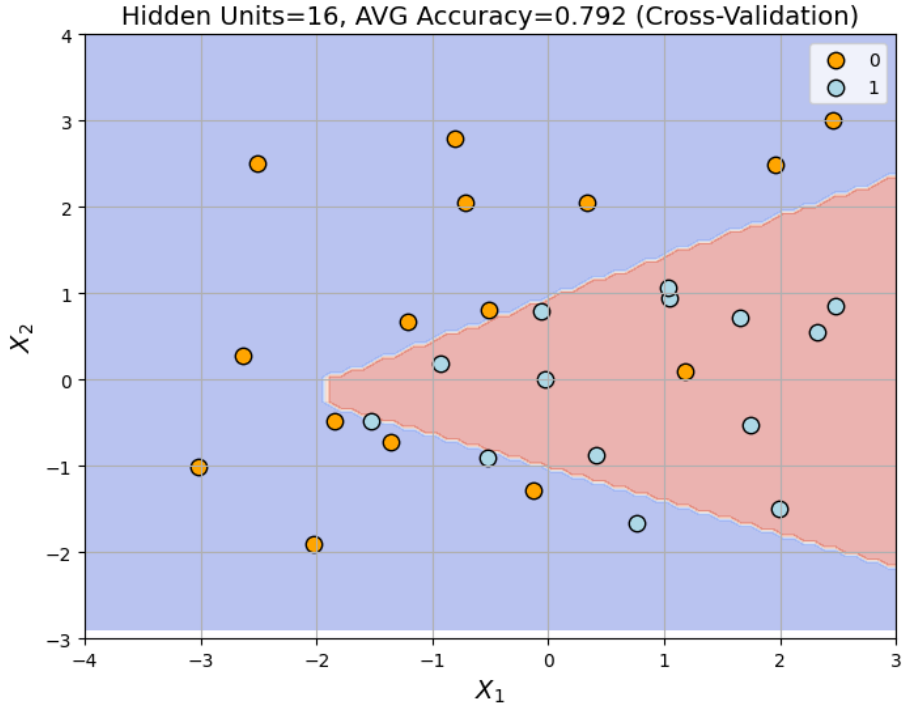
Shrink Weights → Better Generalisation



- Hidden Units: 16
- $\text{MIN}(\boldsymbol{W}) = -59.63, \text{MAX}(\boldsymbol{W}) = 50.71$
- Complex Decision Boundary
- Low Validation Accuracy (0.625)

$\lambda = 0.09$

- Hidden Units: 16
- $\text{MIN}(\boldsymbol{W}) = -3.10, \text{MAX}(\boldsymbol{W}) = 2.72$
- Simpler Decision Boundary
- High Validation Accuracy (0.792)



Linear Regression: California Housing Dataset: Model Interpretation

- Standardisation ensures features with different units and scales are in the same scale and unit. This allows interpretation of model coefficients, in other words standardization enables features of importance>
- 8 features are median income, housing age, average rooms and bedrooms per household, block population, average occupancy, and geographical coordinates (latitude and longitude) of the block group.

$$\vec{w}^T = \begin{bmatrix} 0.826 & (\$100\text{k}) \\ 0.117 & (\$100\text{k}) \\ -0.248 & (\$100\text{k}) \\ 0.290 & (\$100\text{k}) \\ -0.008 & (\$100\text{k}) \\ -0.030 & (\$100\text{k}) \\ -0.900 & (\$100\text{k}) \\ -0.870 & (\$100\text{k}) \end{bmatrix}$$

(Excl. Intercept Term)

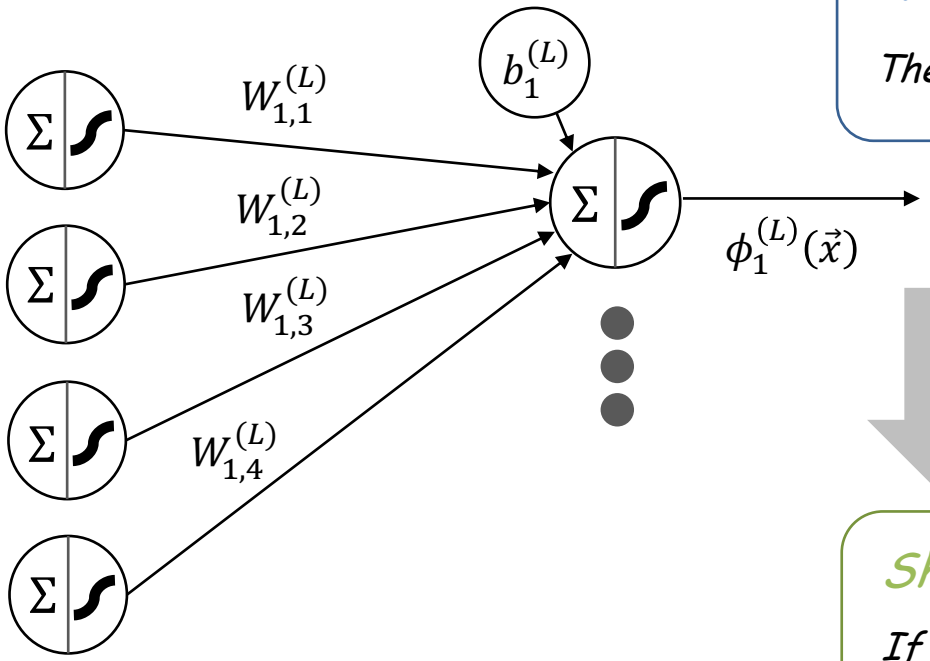
Q: Which features matter most in this model?

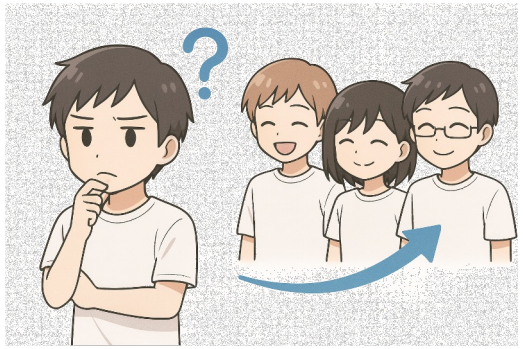
A: Location (latitude / longitude), median income comes next.

Q: How do we know that?

A: Coefficients. The bigger absolute value, the bigger the impact.

Chain-of-Thought: Dropout Intuition





Weight Size → Feature Influence

The larger $\left|W_{1,j}^{(L)}\right|$ is, the more the model leans on feature $\phi_j^{(L-1)}$.

Heavy Reliance → Over-Fitting

Big weights can lock on to quirks and noise that appear only in the training set.

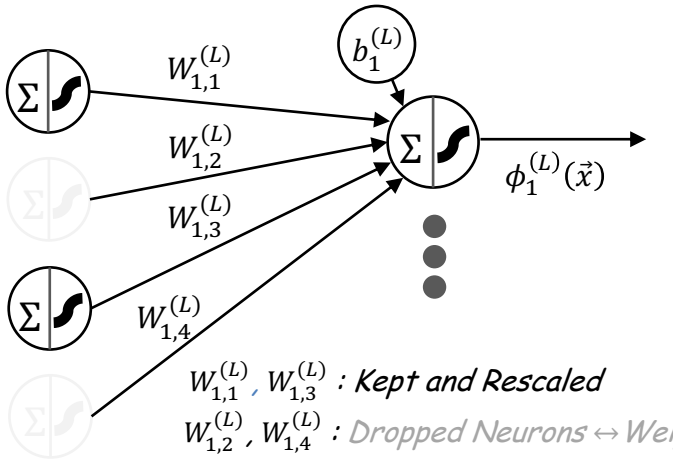
Shared Reliance → Small Weights

If the network cannot depend on any single feature, each weight stays small.

Small Weights → Less Overfitting

A weight profile with mostly modest numbers generalises better to new data.

Training Iterations (t)

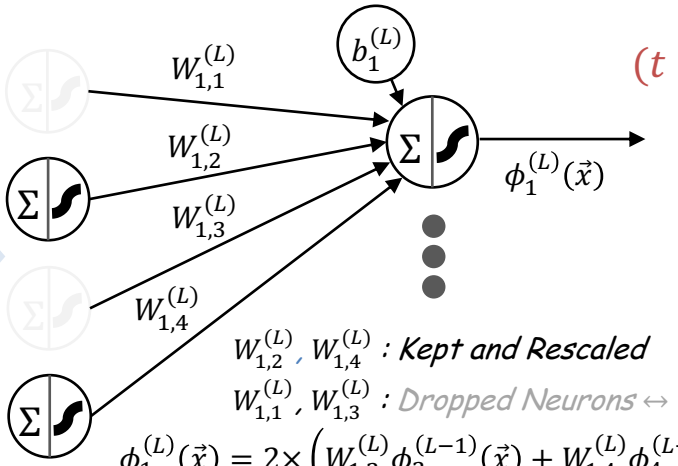


(t)

$W_{1,1}^{(L)}$, $W_{1,3}^{(L)}$: *Kept and Rescaled*

$W_{1,2}^{(L)}$, $W_{1,4}^{(L)}$: *Dropped Neurons ↔ Weight x 0*

$$\phi_1^{(L)}(\vec{x}) = 2\times\left(W_{1,1}^{(L)}\phi_1^{(L-1)}(\vec{x}) + W_{1,3}^{(L)}\phi_3^{(L-1)}(\vec{x})\right) + b_1^{(L)}$$



$(t+1)$

$W_{1,2}^{(L)}$, $W_{1,4}^{(L)}$: *Kept and Rescaled*

$W_{1,1}^{(L)}$, $W_{1,3}^{(L)}$: *Dropped Neurons ↔ Weight x 0*

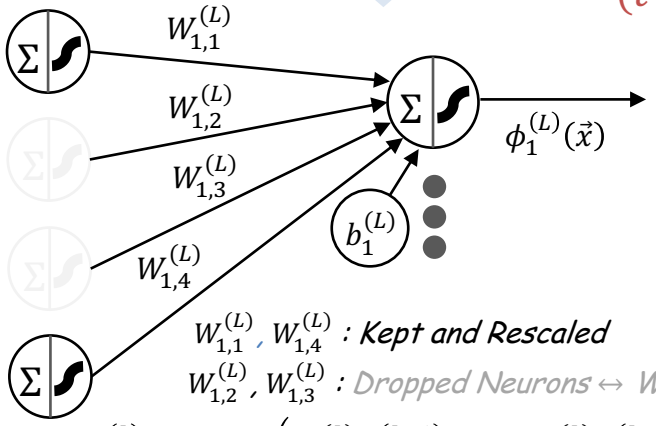
$$\phi_1^{(L)}(\vec{x}) = 2\times\left(W_{1,2}^{(L)}\phi_2^{(L-1)}(\vec{x}) + W_{1,4}^{(L)}\phi_4^{(L-1)}(\vec{x})\right) + b_1^{(L)}$$

New Mask per Batch

Each mini-batch uses a fresh keep or drop pattern.

New Connections x (Rate)$^{-1}$

Surviving weights are rescaled so the layer's expected output stays unchanged.



$(t+2)$

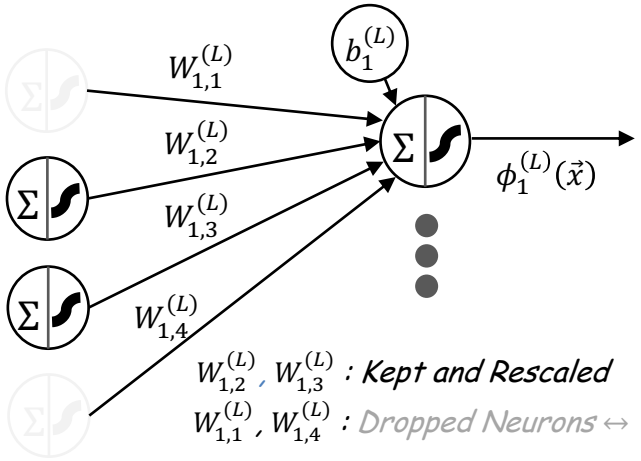
$W_{1,1}^{(L)}$, $W_{1,4}^{(L)}$: *Kept and Rescaled*

$W_{1,2}^{(L)}$, $W_{1,3}^{(L)}$: *Dropped Neurons ↔ Weight x 0*

$$\phi_1^{(L)}(\vec{x}) = 2\times\left(W_{1,1}^{(L)}\phi_1^{(L-1)}(\vec{x}) + W_{1,4}^{(L)}\phi_4^{(L-1)}(\vec{x})\right) + b_1^{(L)}$$

Dropout OFF at Inference

At test-time we run the full network; no units are dropped.



$(t+3)$

$W_{1,2}^{(L)}$, $W_{1,3}^{(L)}$: *Kept and Rescaled*

$W_{1,1}^{(L)}$, $W_{1,4}^{(L)}$: *Dropped Neurons ↔ Weight x 0*

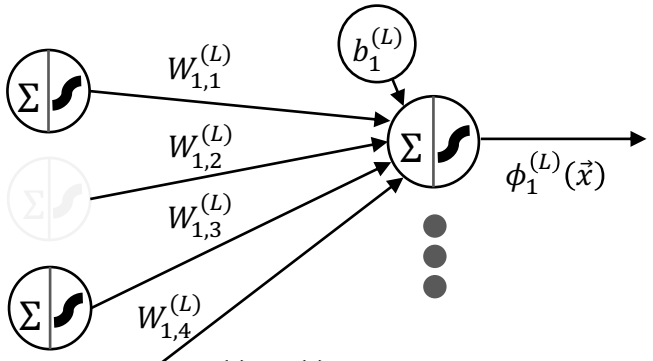
$$\phi_1^{(L)}(\vec{x}) = 2\times\left(W_{1,2}^{(L)}\phi_2^{(L-1)}(\vec{x}) + W_{1,3}^{(L)}\phi_3^{(L-1)}(\vec{x})\right) + b_1^{(L)}$$

Outcome → Less Over-Fitting

Sharing the load forces weights to stay small, leading to better generalization.

Dropout: Inferencing

Training



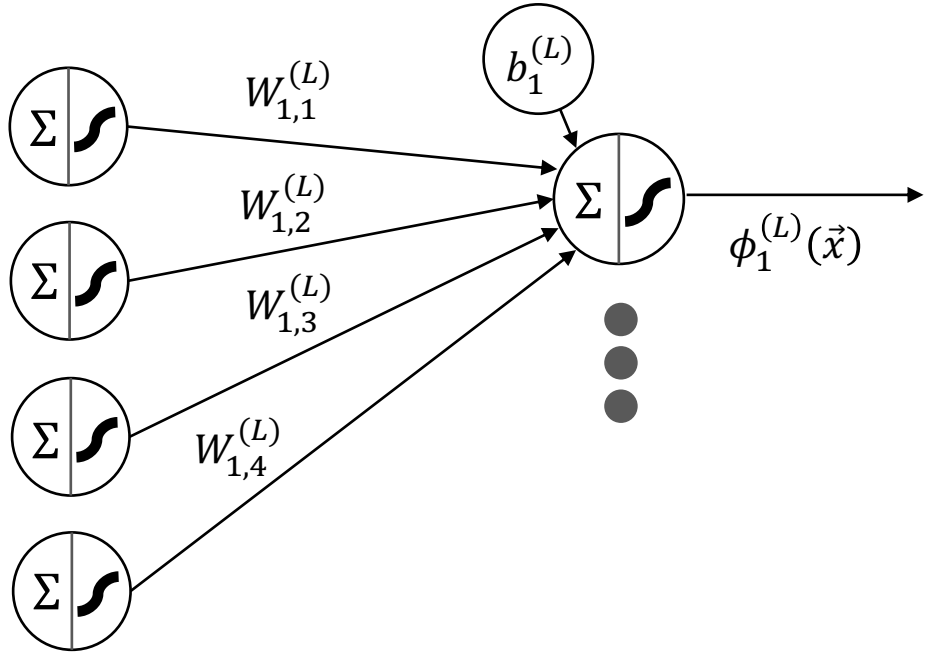
$W_{1,1}^{(L)}$, $W_{1,3}^{(L)}$: *Kept and Rescaled*

$W_{1,2}^{(L)}$, $W_{1,4}^{(L)}$: *Dropped Neurons ↔ Weight x 0*

$$\phi_1^{(L)}(\vec{x}) = 2\times\left(W_{1,1}^{(L)}\phi_1^{(L-1)}(\vec{x}) + W_{1,3}^{(L)}\phi_3^{(L-1)}(\vec{x})\right) + b_1^{(L)}$$

$$\frac{1}{1-\text{Drop Rate }(0.5)}$$

Run-Time (or Evaluation)



Q: When we add dropout, do we also need a special loss function?

A: No. The loss stays exactly the same; dropout only changes the forward pass during training, not the loss definition.

Inference: Dropout OFF — Full Network, Deterministic Output.

$$\phi_1^{(L)}(\vec{x}) = W_{1,1}^{(L)}\phi_1^{(L-1)}(\vec{x}) + W_{1,2}^{(L)}\phi_2^{(L-1)}(\vec{x}) + W_{1,3}^{(L)}\phi_3^{(L-1)}(\vec{x}) + W_{1,4}^{(L)}\phi_4^{(L-1)}(\vec{x}) + b_1^{(L)}$$

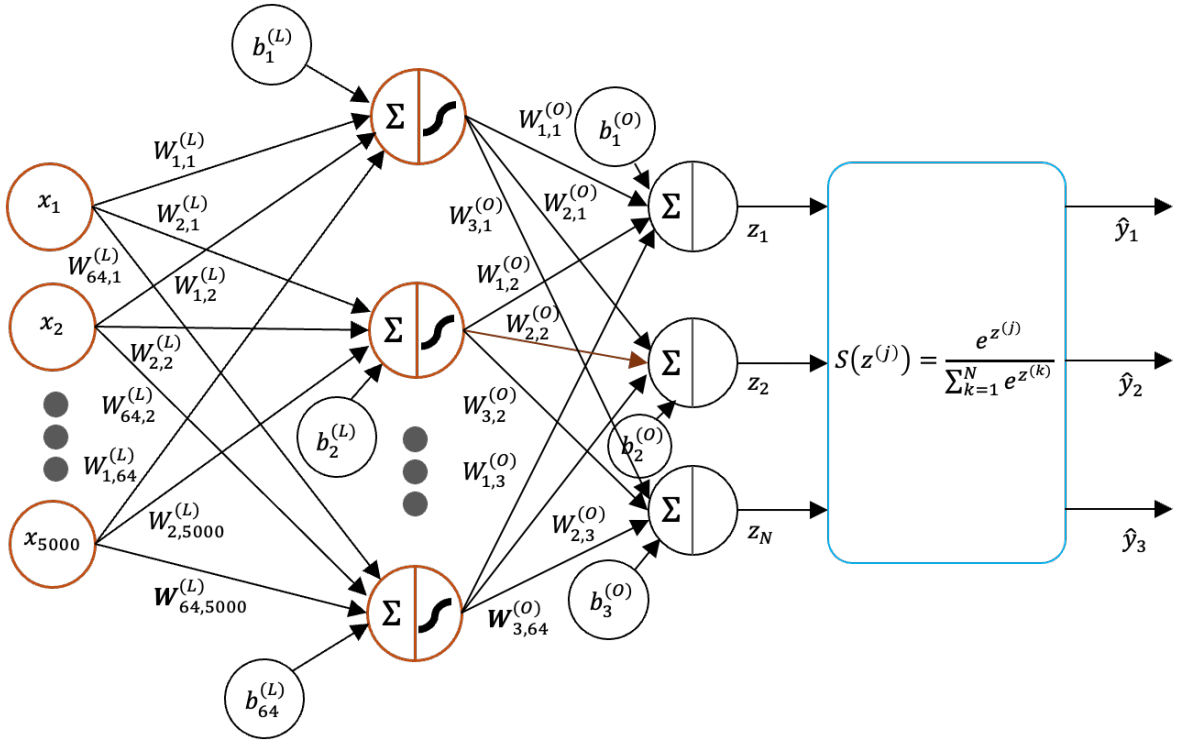
US Airline Sentiment Dataset

Key Facts	Details
Size	≈ 14.6 k English-language tweets (collected 2013-2015).
Target Label	airline_sentiment ∈ { negative (≈63 %), neutral (≈21 %), positive (≈16 %) }.
Airlines Covered	American, United, Southwest, Delta, US Airways, Virgin America.
Main Text Field	text – the raw tweet.
Auxiliary Fields	Tweet ID, time stamp, user handle, airline name, sentiment confidence, tweet latitude/longitude, etc

```
@united delayed again. Sat on the tarmac for an hour,
missed my connection, customer service no help.
```

```
Huge thanks to @SouthwestAir for getting me home
early & with a smile. You guys always deliver!
```

Tensorflow Code Snippet



Dropout randomly turns neurons off during the training.

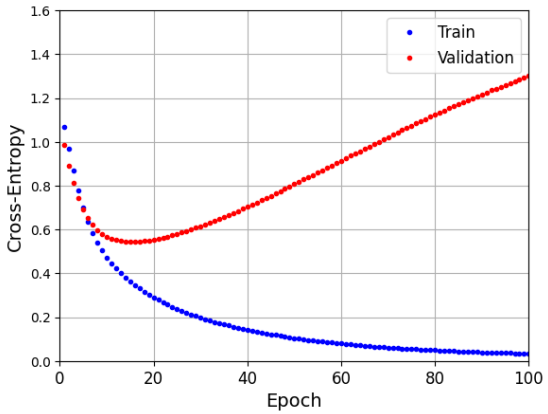
```
@united delayed again. Sat on the tarmac for an hour,
missed my connection, customer service no help.
```

```
Huge thanks to @SouthwestAir for getting me home
early & with a smile. You guys always deliver!
```

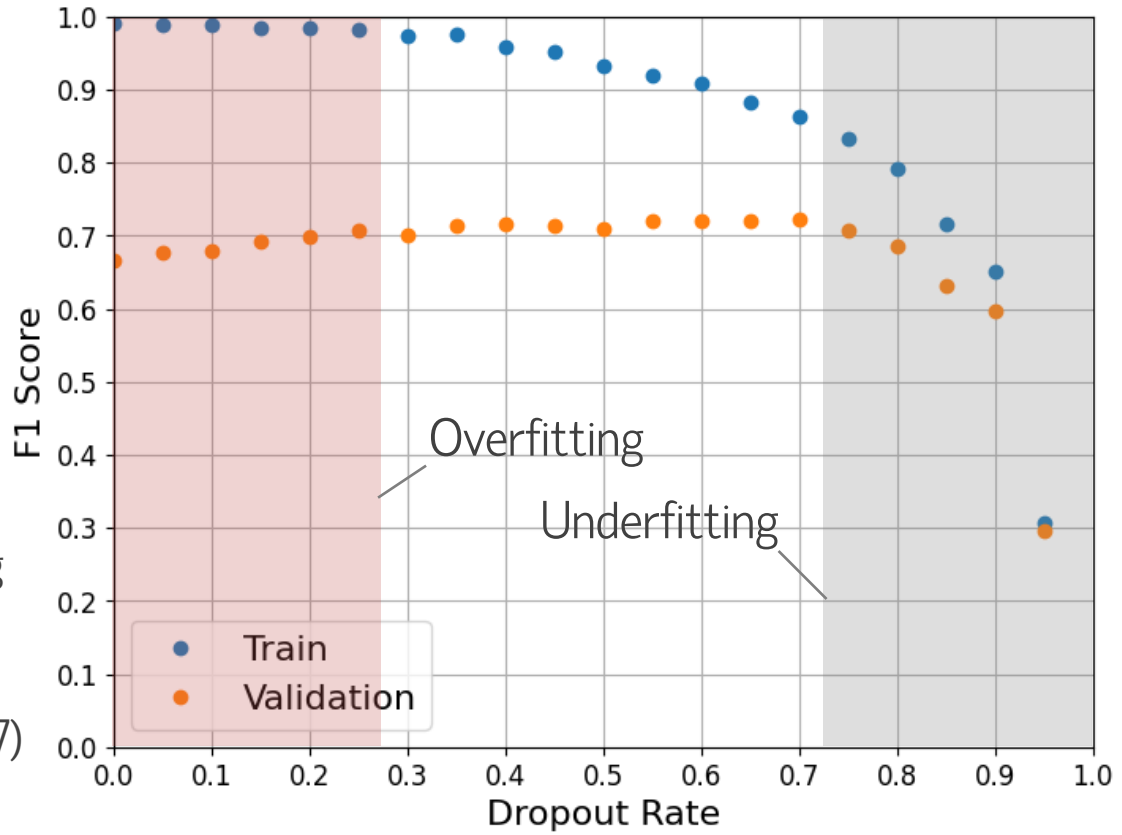
```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense, Dropout
import tensorflow_addons as tfa

num_inputs = 5000       # input dimension
num_hidden_units = 64   # hidden units
num_outputs = 3         # output dimension
drop_rate = 0.5         # 0.5 ⇒ keep-prob p = 0.5

f1 = tfa.metrics.F1Score(num_classes=num_outputs, average="macro")

model = Sequential([
    Dropout(rate=drop_rate, input_shape=(num_inputs,)),

    Dense(num_hidden_units,
          activation="sigmoid"),

    Dropout(rate=drop_rate),

    Dense(num_outputs, activation="softmax")
])

model.compile(optimizer=”sgd",
              loss=”categorical_crossentropy",
              metrics=[f1])

history = model.fit(X_train, y_train,
                    batch_size=512,
                    epochs=100,
                    validation_data=(X_val, y_val))
```

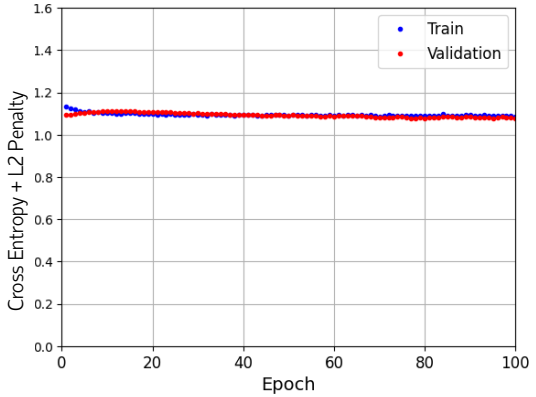
Overfitting vs Underfitting

- Low-dropout region (< ≈ 0.30) – Almost every unit is kept, the network can depend on a few dominant features. Training F1 stays near 1.0 while validation levels off around 0.70, showing classic over-fitting.
- High-dropout region (> ≈ 0.75) – So many activations are dropped that the model sees only fragments of each example. Both training and validation F1 fall to ~0.30, a sign of under-fitting caused by too little signal.



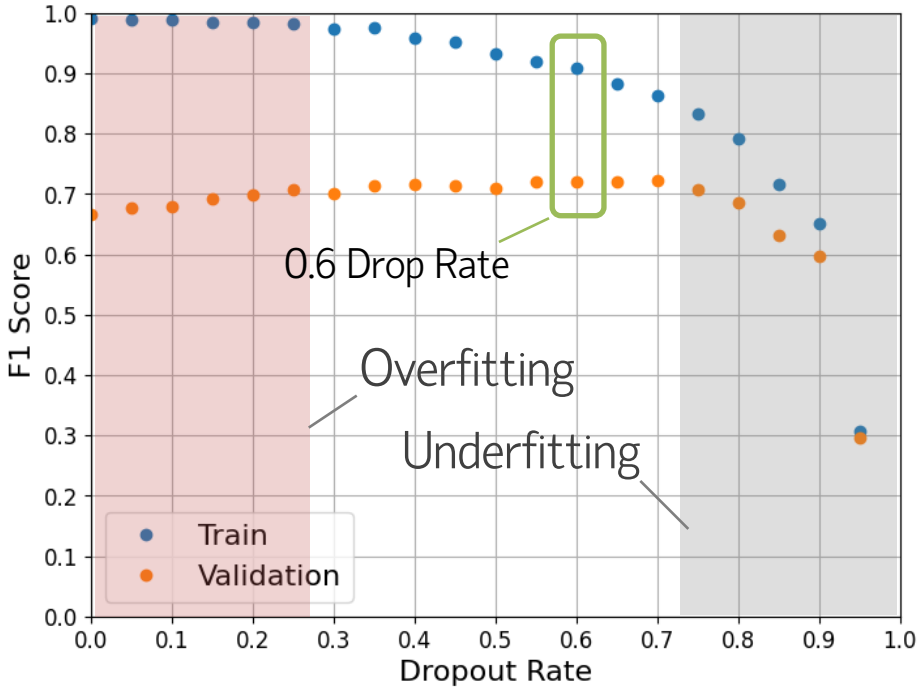
- No Feature Hedging
- $\|W\|_2 = 44.36$
- Getting Better at Memorising Training Data
- High Train F1 Score (0.99)
- Low Validation F1 Score (0.67)

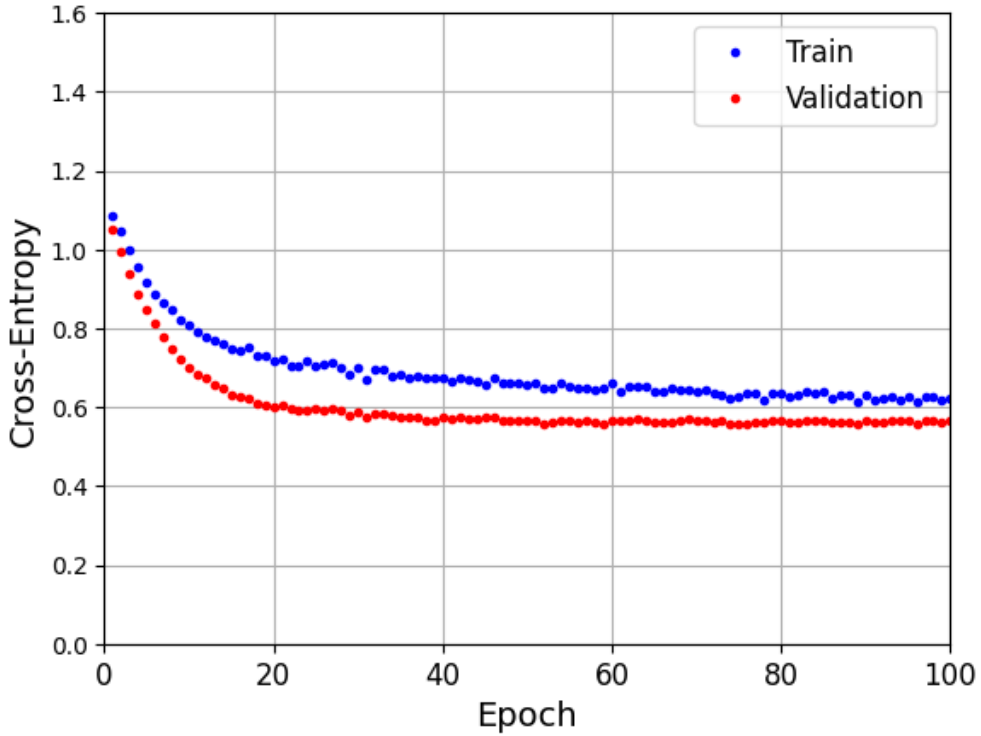




- High Hedge Ratio (No Reliance on Specific Features)
- $\|W\|_2 = 14.90$
- Unable to Learn Intricated Relationships
- Low F1 Scores on both Train (0.39) and Validation (0.39) Data

Optimal Model





- Moderately Dropout Rate
- $\|W\|_2 = 36.13$
- Highest F1 Score (0.72) on Validation Data
- Train F1 Score (0.90)

Dropout had brought $\|W\|_2$ down, but still high in comparison to L2 regularisation.

Q: Eh? How did the model mange not to overfit the data?

A: Dropout builds an implicit ensemble of thinned sub-nets whose averaged vote at test-time smooths predictions, while the random "zeros" add training noise that further regularises the model—together they keep it from memorising spurious patterns.

Dropout ≈ Bagging + Noise + Implicit Weight-Decay

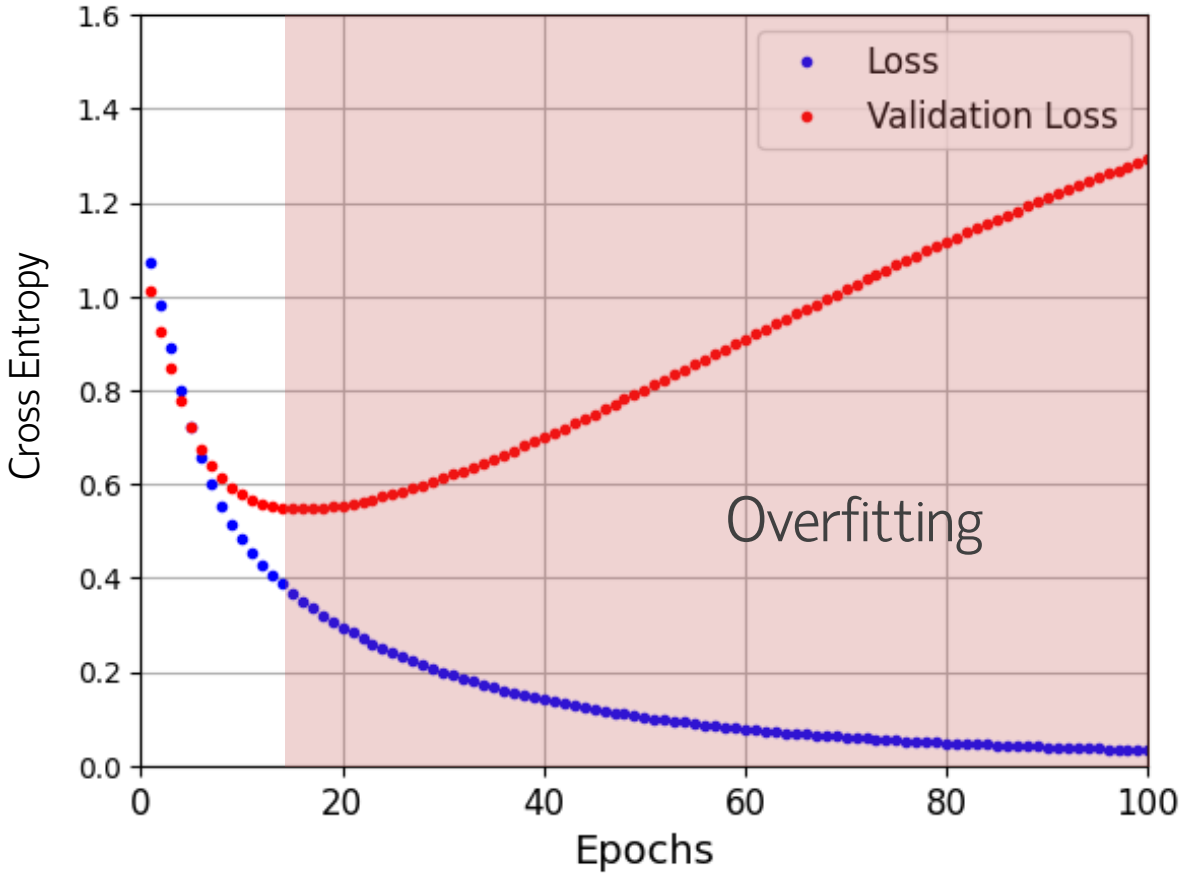
Why Dropout Helps

What Dropout Does	Intuition	Practical Benefit
Trains an implicit ensemble	Each mini-batch sees a different thinned sub-net. At test-time all units are active, so the final prediction is the *average vote* of thousands of sub-nets.	Ensemble averaging reduces variance and smooths decision boundaries without storing multiple models.
Breaks up feature co-adaptation	Because an input can disappear at any step, a unit can't rely on one "partner" feature; it must find *redundant* cues.	Redundancy makes the representation more robust to noise or missing values at inference.
Acts like noise injection	Zeroing activations adds unbiased noise to the forward pass; back-prop sees a noisy gradient.	Noise has a regularising effect similar to data augmentation, encouraging smoother functions.
Produces an implicit L2 penalty	The expectation of the dropped network equals the full network with weights scaled by (1-p). Algebraically that's equivalent to weight-decay on the original parameters.	Explains why dropout often keeps weights small even when you don't add explicit L2.
Encourages sparse, distributed codes	Units can't assume they will always fire together, so activations spread across wider groups.	Useful in NLP/vision: downstream tasks can pick up many weak but complementary signals.

Where to Use Dropout (and Why)

Layer Position	Apply Dropout?	Why / Why Not	L2 Weight-Decay?
Input (feature) layer	*Sometimes* – "feature dropout," helpful for very high-dim TF-IDF, images with heavy noise, etc.	Stops the model from leaning on any single raw feature.	Always safe; acts on weights, not raw inputs.
Hidden fully-connected layers (including **last hidden → output input**)	**Yes – default place.** Insert right after the hidden activation, e.g. Dense → ReLU → Dropout(p) even for the last hidden layer that feeds the classifier head.	Breaks co-adaptations and works like model averaging; biggest regularisation gain in MLPs.	Common; complements dropout by shrinking large weights.
Convolutional layers	Mixed. Prefer **SpatialDropout / DropBlock** that drops entire channels or blocks; plain dropout can damage spatial coherence.	Keeps spatial structure while still regularising.	Standard on conv kernels.
Recurrent layers	Use **locked / variational dropout** between stacked RNN/LSTM layers or on inputs; avoid time-step-wise dropout inside the recurrence.	Locked mask preserves temporal consistency.	L2 on recurrent weights is less disruptive than dropout and routinely used.
Immediately after batch-norm	**Skip.** Batch-norm already injects noise and reduces co-adaptation; dropout here gives little extra benefit and can slow training.	—	L2 still recommended.
Output layer itself (after Dense(output) or after Softmax / Sigmoid)	**No.** We want deterministic logits / probabilities at inference; dropping them adds noise but no useful regularisation.	Test-time behaviour mismatches the noisy training logits.	Harmless and common.

Training History

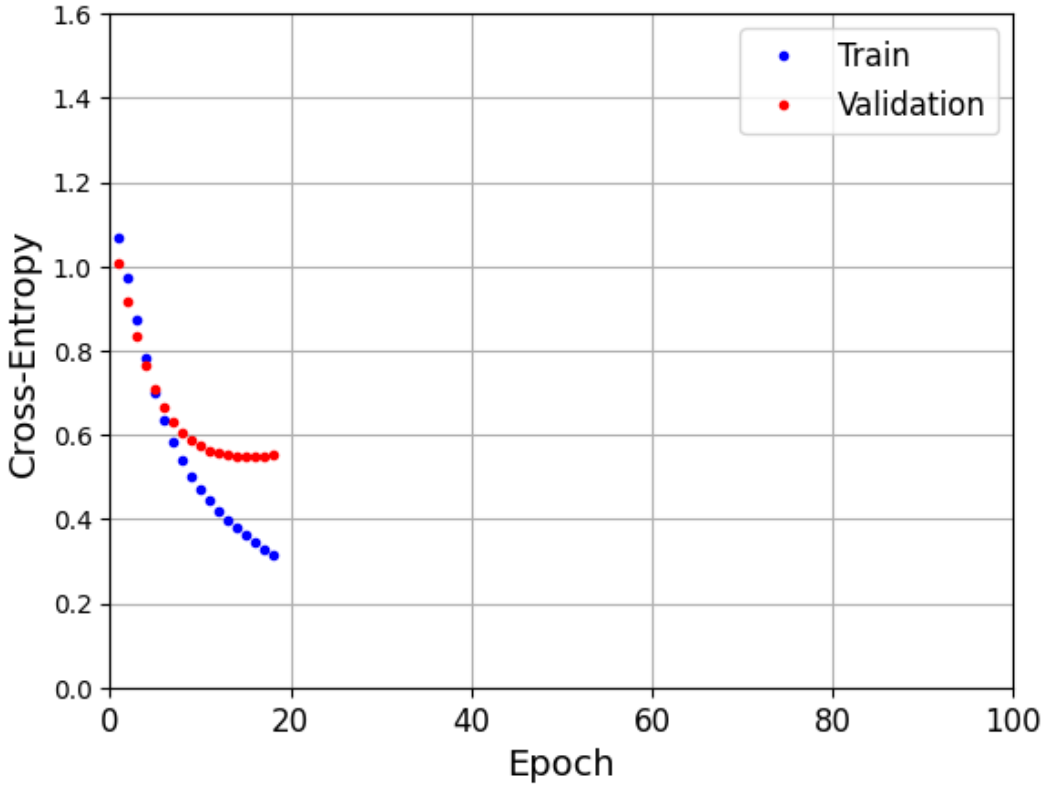


The validation loss reached its minimum value of 0.54 at epoch 15, after which the model began to over-fit.

Q: If we know validation loss bottoms out at epoch 15, why keep training and let the model over-fit?

*A: We don't have to. In practice we add an "**Early-Stopping**" callback that stops training as soon as the validation loss stops improving and restores the best weights.*

Early Stopping



- F1 Score: 0.73 (Validation)
- Training Stopped: 18 Epochs (→Best at 15 Epochs)

Eh? The performance is even better than when regularising the model.

Why bother to regularise then?

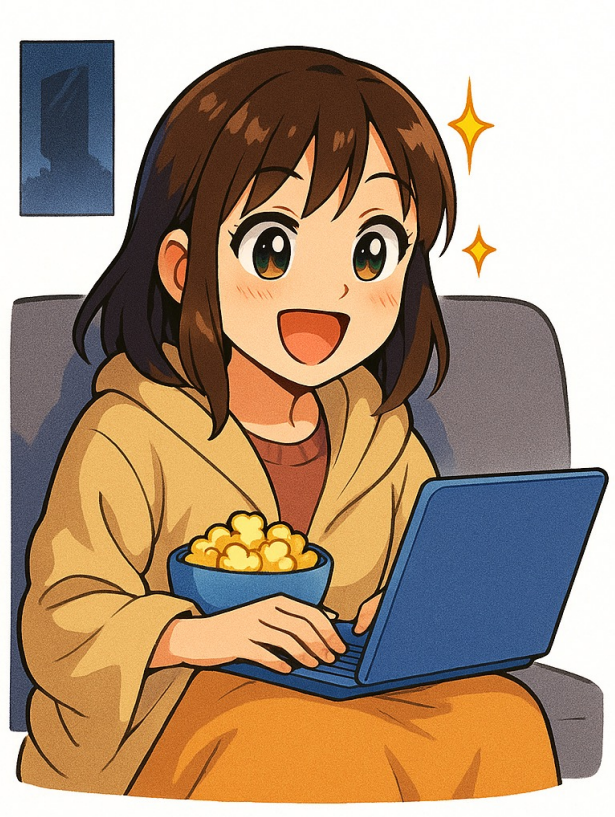
```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense, Dropout
import tensorflow_addons as tfa
from keras.callbacks import EarlyStopping

num_inputs = 5000       # input dimension
num_hidden_units = 64   # hidden units
num_outputs = 3         # output dimension
drop_rate = 0.5         # 0.5 ⇒ keep-prob p = 0.5

f1 = tfa.metrics.F1Score(num_classes=num_outputs, average="macro")

model = Sequential([
    Dense(num_hidden_units,
          activation="sigmoid"),

    Dense(num_outputs, activation="softmax")
])

model.compile(optimizer=”sgd",
              loss=”categorical_crossentropy",
              metrics=[f1])

early_stop = EarlyStopping(monitor=‘val_loss’,
                           restore_best_weights=True, patience=3)

history = baseline.fit(X_train, y_train,
                       batch_size=512,
                       epochs=100,
                       validation_data=(X_val, y_val),
                       callbacks=[early_stop])
```

IMDB Movie Review Dataset

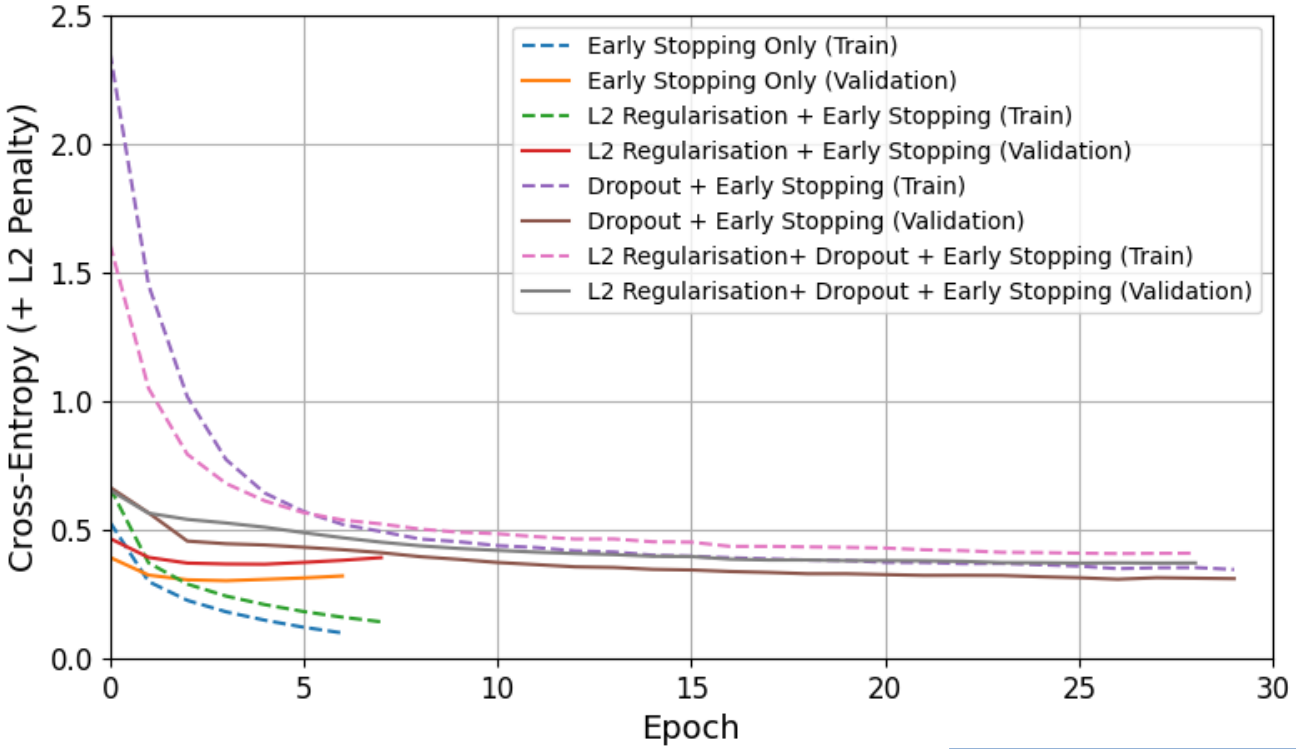
Key Facts	Details
Size	50,000 English-language movie reviews (25k train, 25k test)
Target Label	Binary sentiment ∈ { positive (★≥7), negative (★≤4) }
Review Sources	IMDb user reviews (movies from various genres and years)
Main Text Field	review – raw HTML-stripped user review text
Auxiliary Fields	review ID, original score (optional), train/test split, etc.

```
This was 2 hours of my life I'll never get back.
Bad acting, zero plot, and the dialogue was
cringeworthy.
```

```
Absolutely stunning! Beautifully directed with
powerful performances.
A must-watch for anyone who loves meaningful cinema.
```



Why / Why Not

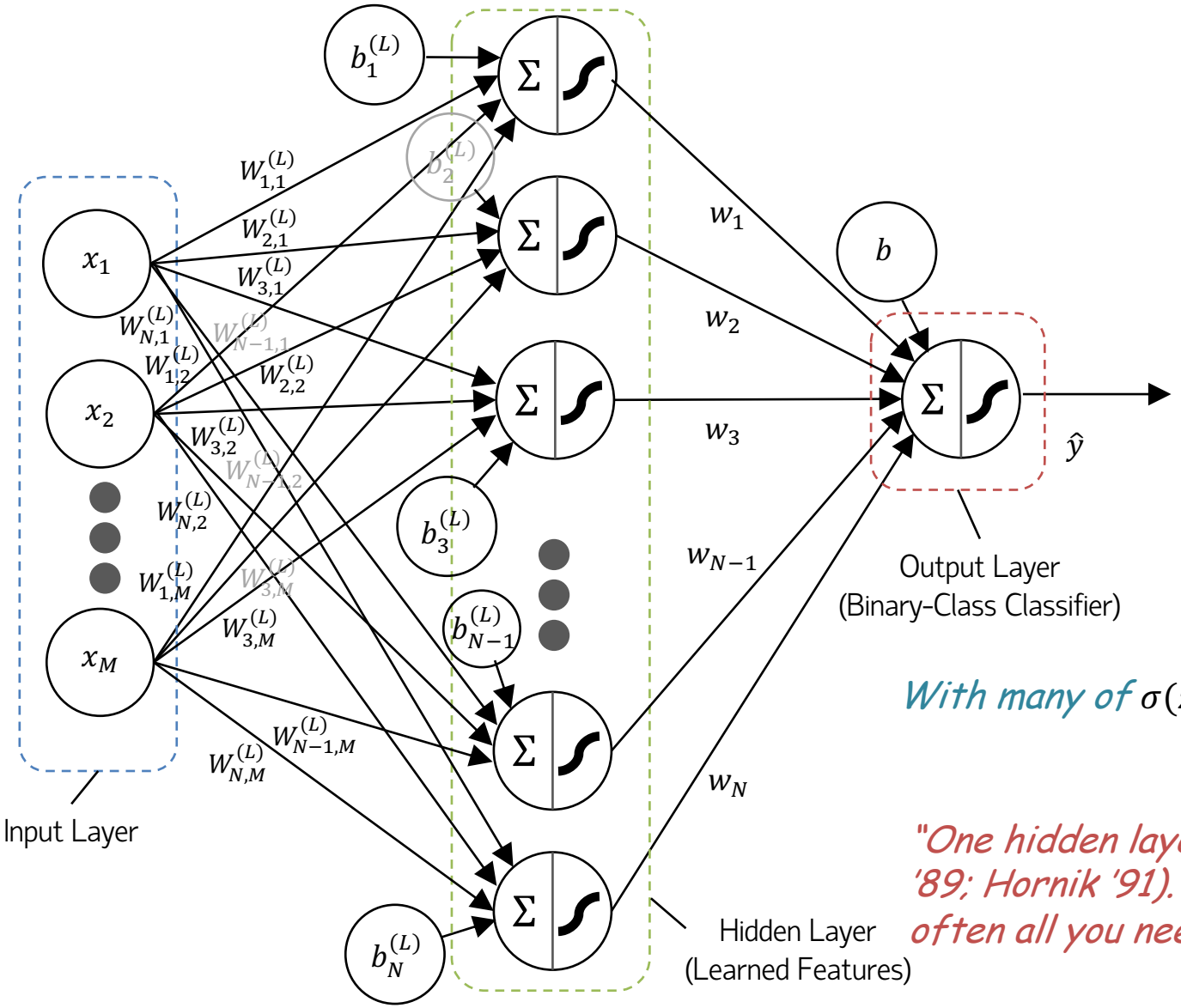


Early-stopping alone can be surprisingly strong on small, shallow text models.

But in wider/deeper nets, higher λ, larger dropout, or noisier data, the combined approach usually widens the gap well beyond the +0.01 we saw.

Model	F1 Score	MIN($\boldsymbol{W}$)	MAX($\boldsymbol{W}$)	$\|\boldsymbol{W}\|_2$
Early Stopping Only	0.876	-2.36e-01	2.32e-01	16.27
L2 Regularisation + Early Stopping	0.875	-2.37e-01	2.45e-01	13.97
Dropout + Early Stopping	0.884	-2.37e-01	2.25e-01	16.92
L2 Regularisation + Dropout + Early Stopping	0.885	-2.27e-01	2.50e-01	14.08

Wide Network



Forward Path:

$$\Phi^{(L)}(\vec{x}) = \sigma\left(\boldsymbol{W}^{(L)} \times \vec{x} + \vec{b}^{(L)}\right)$$

$$\hat{y} = \sigma\left(\vec{w}^T \times \Phi^{(L)}(\vec{x}) + b\right)$$

Learned Features

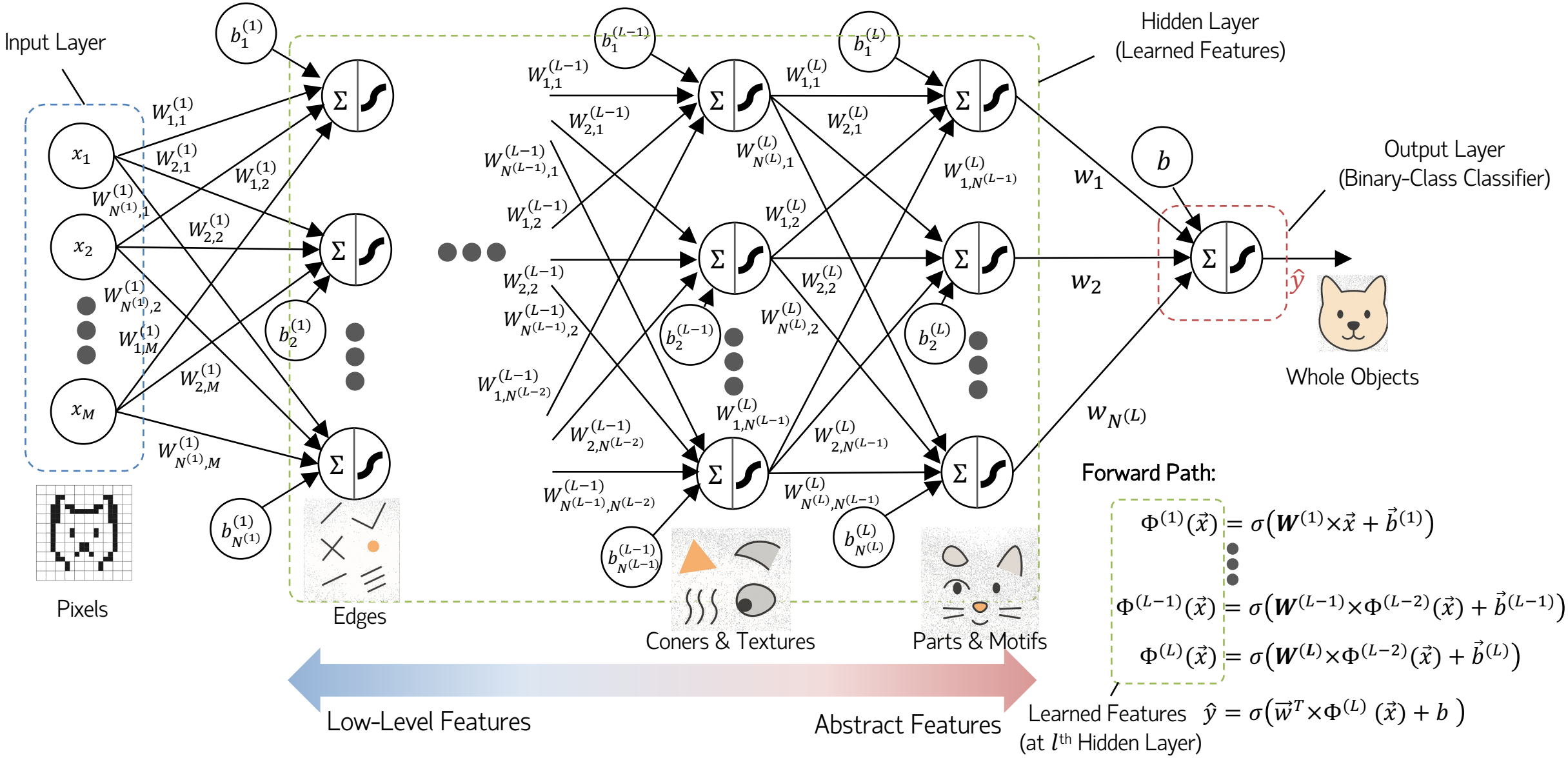
With many of $\sigma(z)$s, we can capture any underlying complexities.

"One hidden layer is already a universal approximator (Cybenko '89; Hornik '91). On tabular data, a single but wide layer is often all you need (Gorishniy et al., 2021)."

Wide Network (cont.)

Finding	Supporting Research & Key Takeaway
A single hidden layer can approximate any continuous function if you give it enough units (i.e. a "wide" network).	Universal-Approximation Theorem—Cybenko (1989) and Hornik et al. (1991) proved that one hidden-layer MLP with a nonlinear activation can approximate any continuous function on a compact domain to arbitrary accuracy, provided the layer is wide enough.
Depth can replace some of that width, but width alone still works—and in practice you often widen when depth is fixed.	Eldan & Shamir (2016) show functions that need exponentially more units in one hidden layer than in a modestly deep net, explaining why "deep *or* wide" is the trade-off. Theoretical papers on the lottery-ticket hypothesis also highlight that very wide nets contain many good (sparse) sub-networks.
For many structured/tabular problems, a single hidden layer that is merely "reasonably wide" already matches deeper architectures.	Large empirical study on tabular data (Gorishniy et al., 2021) finds tuned shallow MLPs (often just 1–2 hidden layers) perform on par with fancier deep models on dozens of real-world datasets. In practice, practitioners start with one hidden layer and scale width first.

Deep Network



$$\Phi^{(1)}(\vec{x}) = \sigma\left(\boldsymbol{W}^{(1)} \times \vec{x} + \vec{b}^{(1)}\right)$$

$$\Phi^{(L-1)}(\vec{x}) = \sigma\left(\boldsymbol{W}^{(L-1)} \times \Phi^{(L-2)}(\vec{x}) + \vec{b}^{(L-1)}\right)$$

$$\Phi^{(L)}(\vec{x}) = \sigma\left(\boldsymbol{W}^{(L)} \times \Phi^{(L-2)}(\vec{x}) + \vec{b}^{(L)}\right)$$

$$\hat{y} = \sigma\left(\vec{w}^{T} \times \Phi^{(L)}(\vec{x}) + b\right)$$

Deep Network (cont.)

Evidence	Key Insight for Deep (Multi-Layer) Networks	Typical Data Regime
Eldan & Shamir 2016; Telgarsky 2016 – certain functions need **exponentially more units in 1-layer nets** than in nets with 2–3 hidden layers.	**Depth can express complex functions with far fewer parameters** than an equally powerful wide single layer.	Any data whose target depends on compositions of simpler patterns.
Krizhevsky et al. 2012 (AlexNet); He et al. 2016 (ResNet) and thousands of follow-ups.	Deep hierarchies learn **low-level → mid-level → object-level features** automatically, giving state-of-the-art results for images, audio, text.	**Unstructured high-dimensional inputs** (images, speech, language, time-series).
Gorishniy et al. 2021; Shwartz-Ziv & Armon 2022 (large tabular benchmark).	For many **structured / tabular** tasks, **1–2 hidden layers** (properly tuned) match or beat deeper MLPs; tree-based methods often stay competitive.	Structured feature tables – banking, insurance, tabular Kaggle sets.
Universal-Approximation Theorem (Cybenko 1989; Hornik 1991) + practical work on "lottery tickets".	A single huge layer **can** approximate anything, but depth finds the same function with **fewer parameters and easier optimisation**.	Helps when model size / compute budget matter.

- Wide + Shallow: Simple and Strong on Tabular Data
- Deep + Moderate Width: Preferred for Images, Audio, Text

On structured data, going deeper than two hidden layers is often over-kill unless the feature engineering is minimal and you have very large training sets.

Summary

- ReLU, Sigmoid, and Tanh act as the non-linear "basis functions" in a neural network. Inserted at the hidden layers, they let the model learn new features from the data and bend the decision surface, so underlying relationships that a straight line could not capture become separable.
- Neural networks learn via a two-step loop. A forward pass pushes the data through every layer and produces a single loss value. A backward pass then works in reverse: the gradient of that loss is calculated once at the output and, by the chain rule, is re-used layer-by-layer to obtain $\partial L / \partial w$ for every weight. Nothing "travels" backwards along the wires—only calculus does. Repeating this cycle with gradient-descent updates steadily drives the loss down on both training and validation data.
- Large weights can indicate memorisation. L2 regularisation shrinks those weights, hence implicitly simplifies the model. Dropout breaks up co-adaptations by randomly silencing units during training, forcing the network to spread the load across many paths and, in turn, learn more robust feature combinations. Early-stopping monitors validation loss and freezes the weights as soon as that loss stops improving and keep the model from overfitting.
- Model capacity should match the task. A single, reasonably wide hidden layer is usually adequate for well-engineered tabular data, whereas moderately deep stacks excel on images, audio, and text. Adding layers beyond what the problem demands often yields little extra benefit while increasing training time and risk of over-fitting.