

Data Science

Lecture 5: Deep Learning Overview



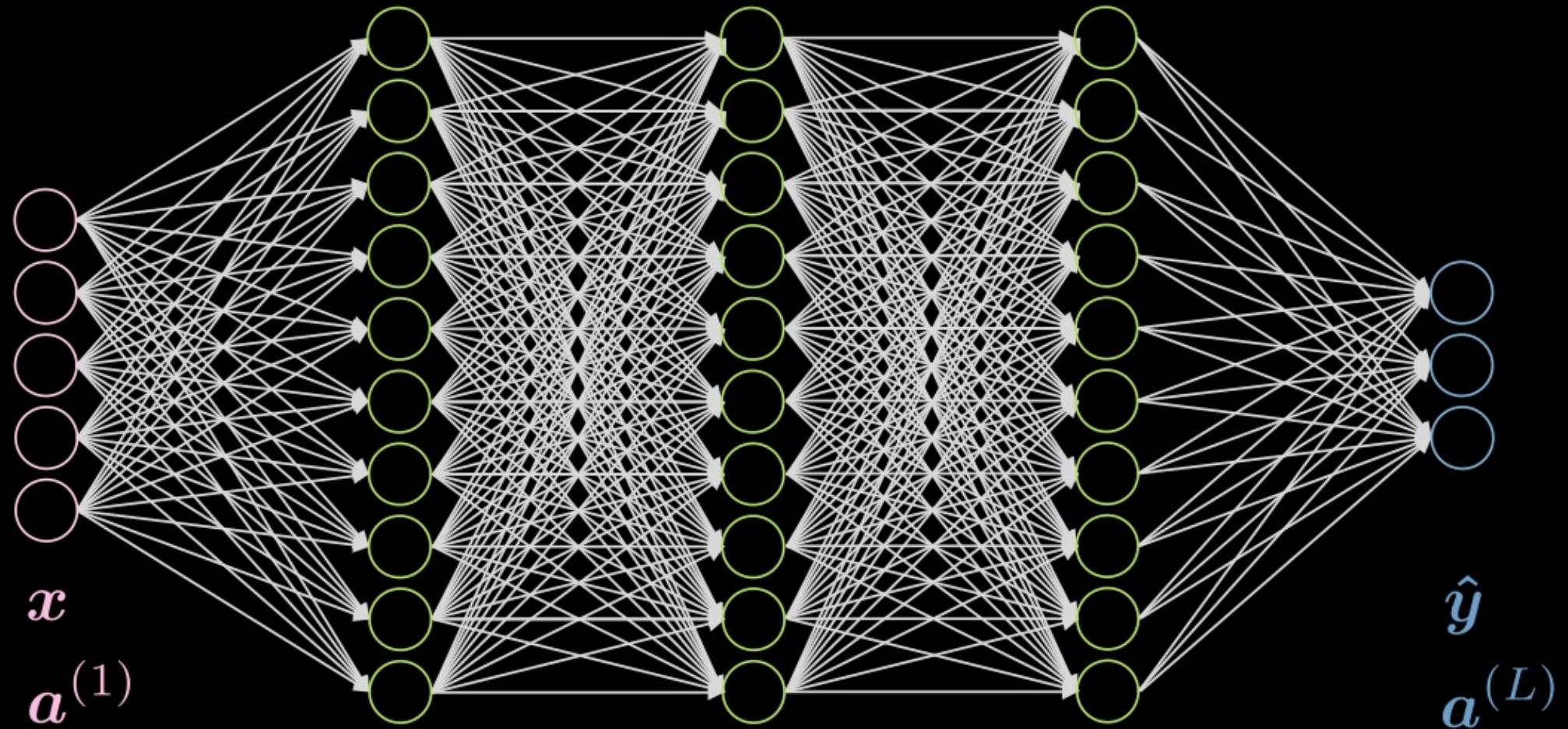
UNIVERSITY
OF AMSTERDAM

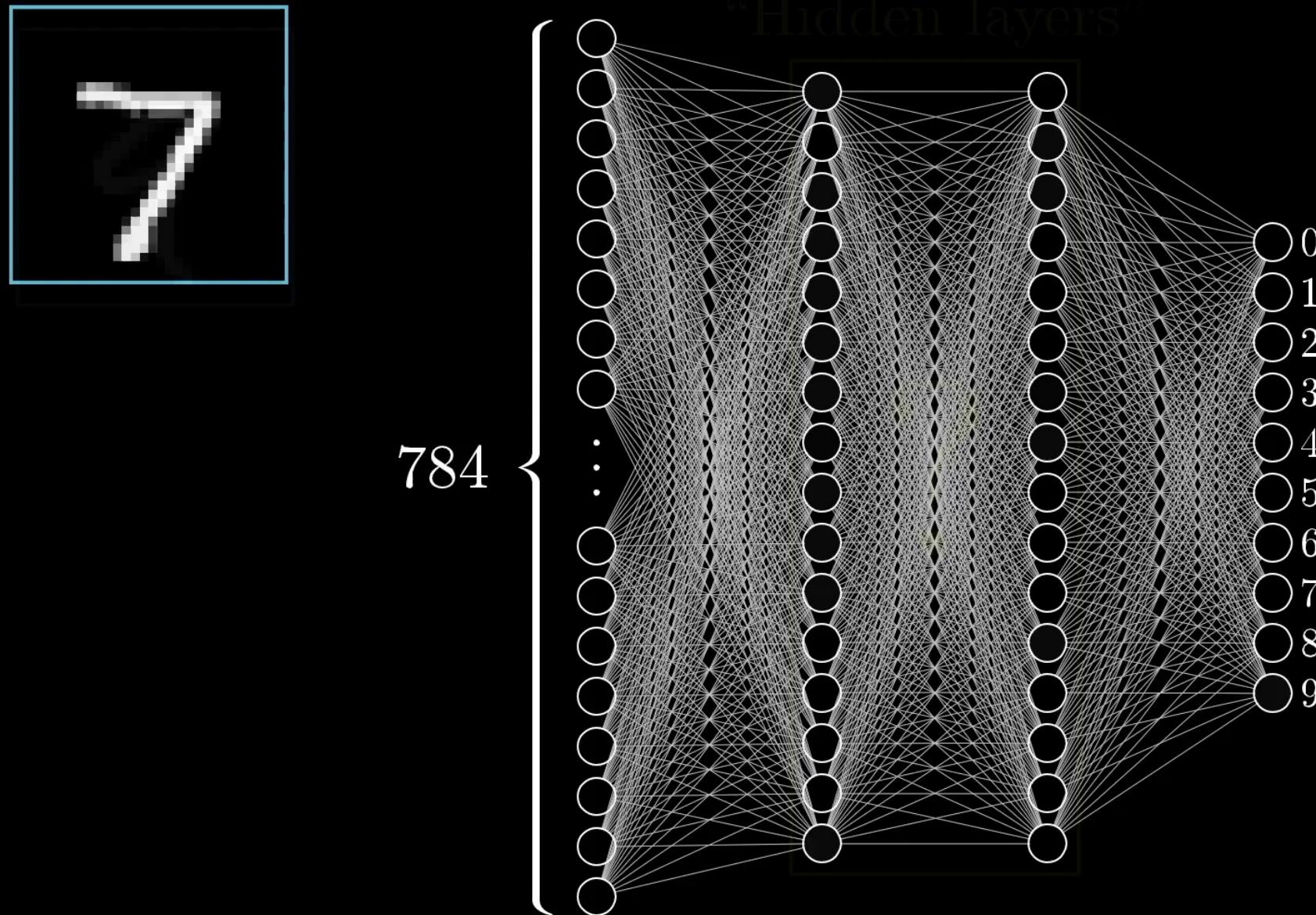
Lecturer: Yen-Chia Hsu

Date: Sep 2025

This lecture will give a high-level overview of
deep learning and artificial neurons.

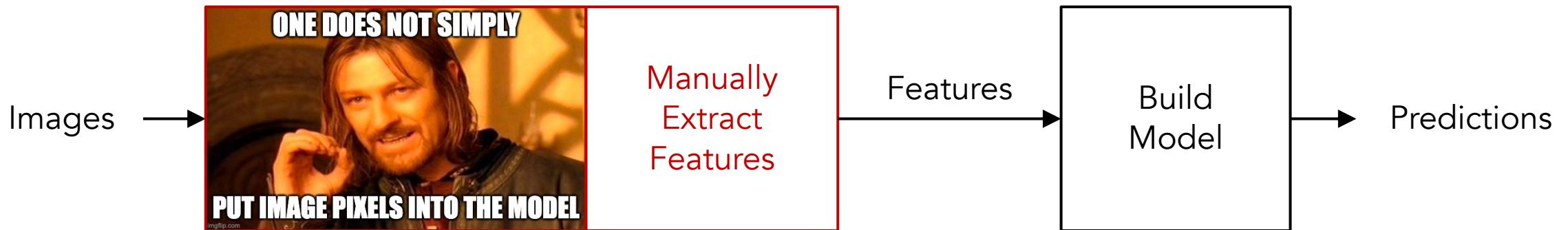
Deep learning is the idea of stacking different types of layers (e.g., **artificial neurons**) to perform very complex tasks. The example below is a deep feedforward neural network.



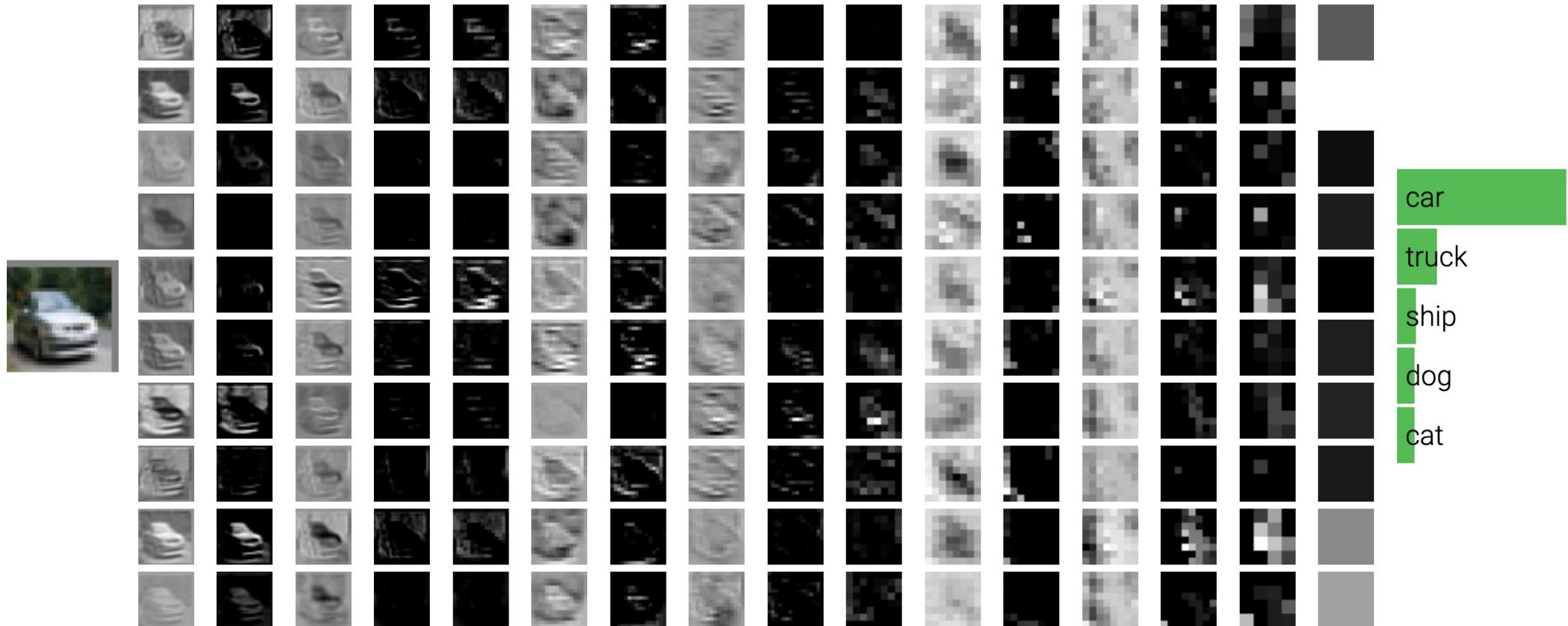


Neurons in the first layer are activated based on the input. Then, neurons are activated layer by layer.

Before the deep learning era, machine learning researchers need to extract features from the data manually. But now, we can delegate feature engineering to the neural net.



Instead of relying on manually crafted features, deep learning models can **learn different representations** from data automatically (i.e., the so-called deep features).



Deep learning models can **extract features automatically** and existed long ago but were not widespread due to the **high demand for computational resources and power**.

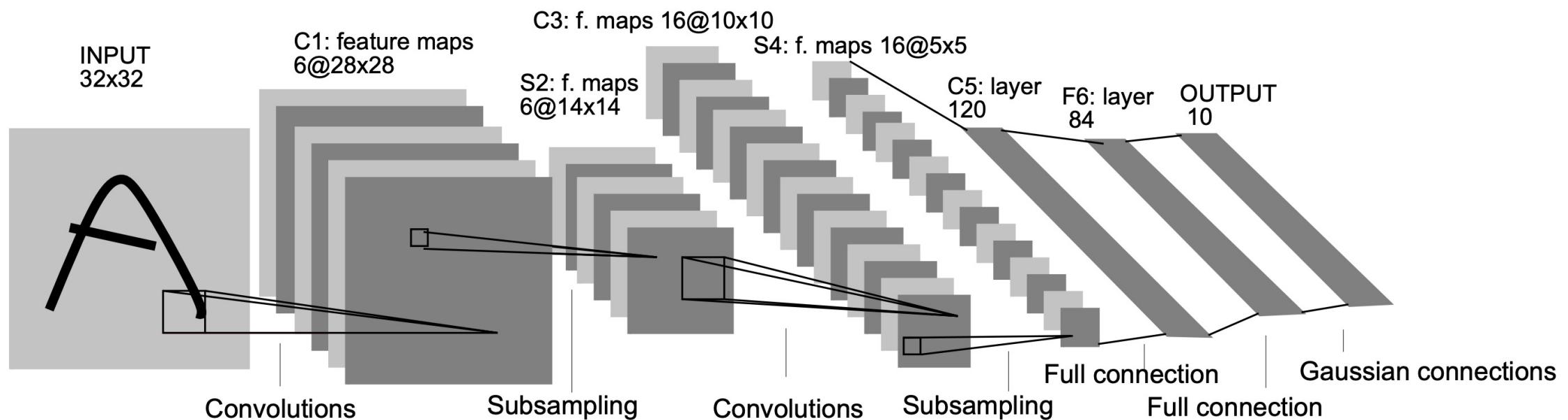


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

In 2012, **AlexNet** showed great improvement in classifying images (ImageNet dataset) using a **Convolutional Neural Network** on GPUs. Then, the deep learning era started.

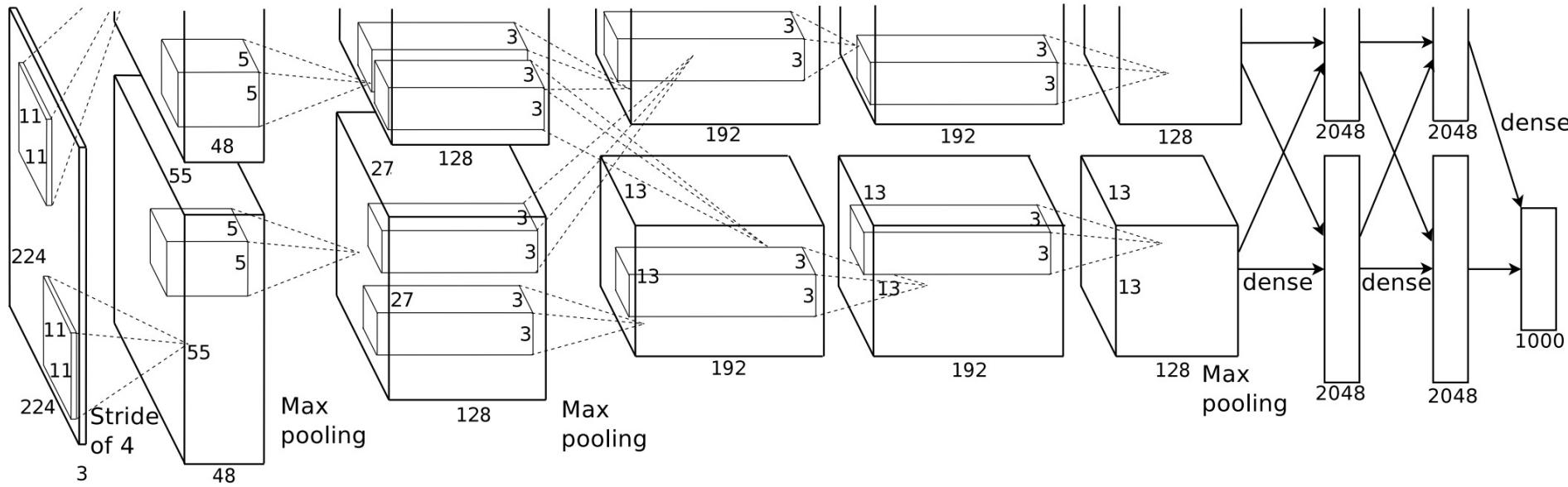
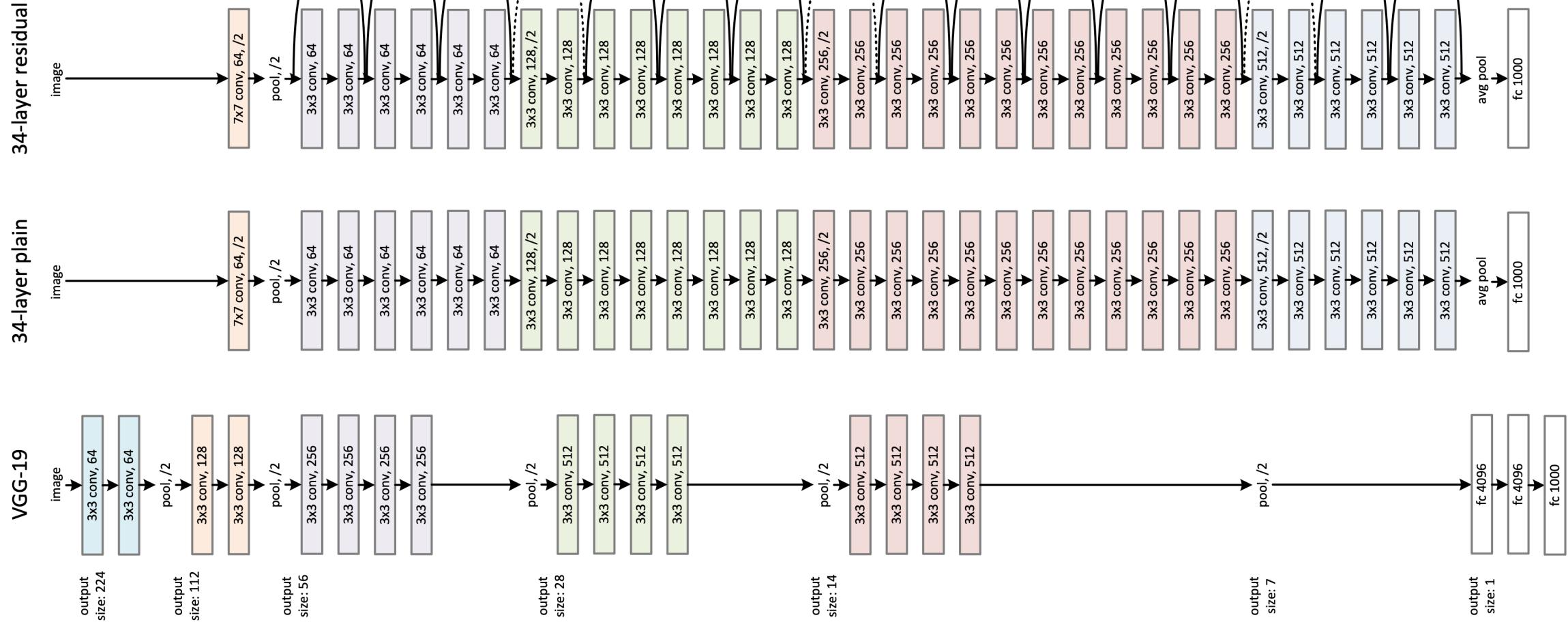


Figure 2: An illustration of the architecture of our CNN, explicitly showing the delineation of responsibilities between the two GPUs. One GPU runs the layer-parts at the top of the figure while the other runs the layer-parts at the bottom. The GPUs communicate only at certain layers. The network's input is 150,528-dimensional, and the number of neurons in the network's remaining layers is given by 253,440–186,624–64,896–64,896–43,264–4096–4096–1000.

Then, deep neural networks went deeper and deeper with different variations.



For sequential data, we can use the **Recurrent Neural Network (RNN)** architecture.

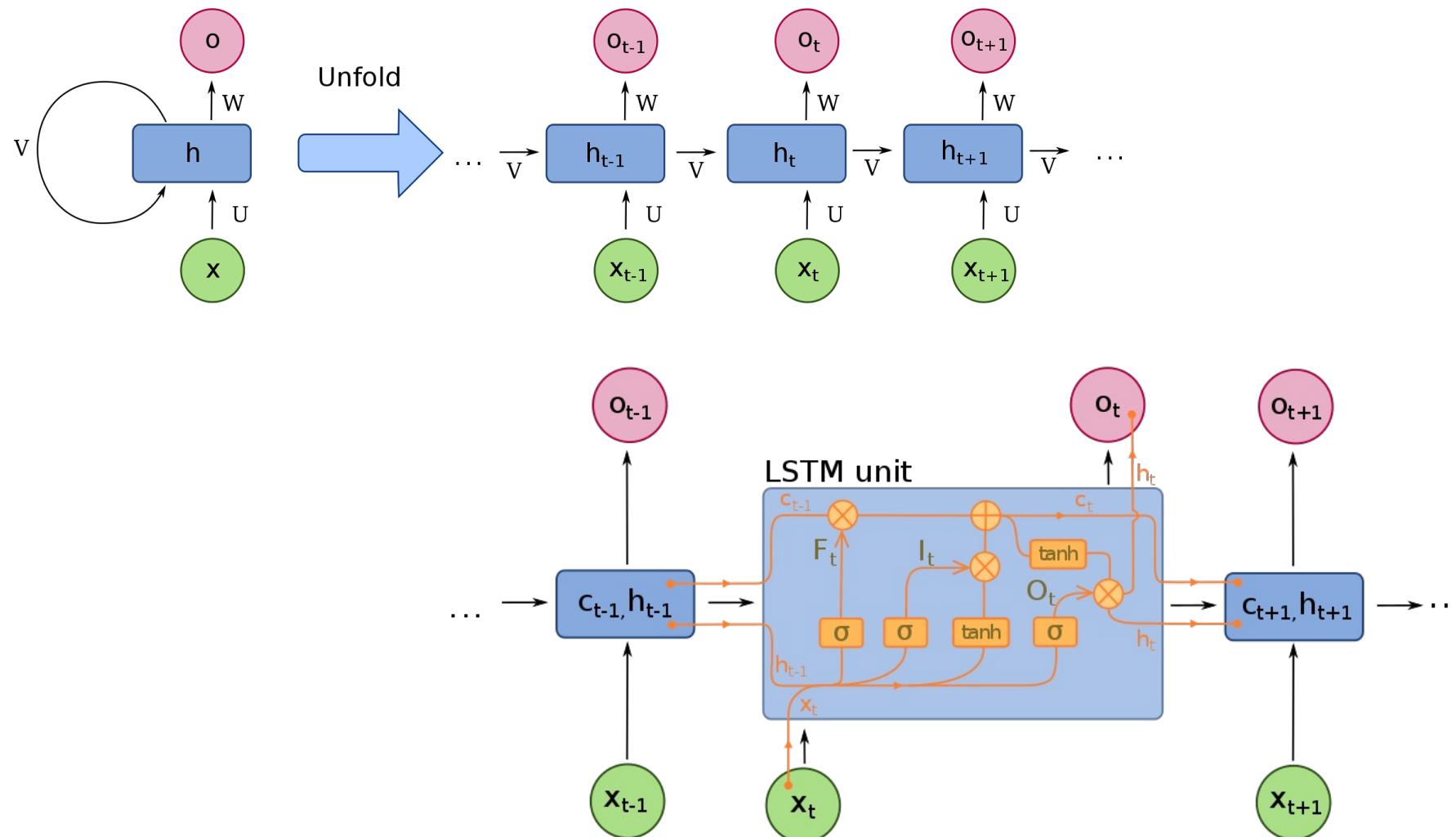
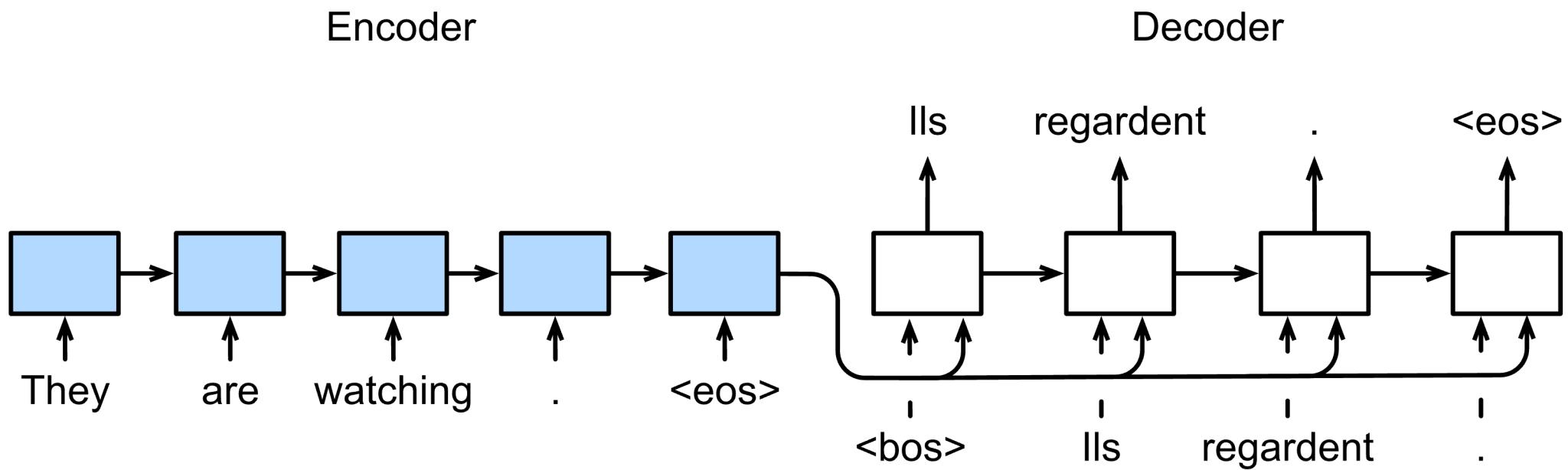


Figure source -- https://en.wikipedia.org/wiki/Recurrent_neural_network

For machine translation, the **sequence-to-sequence model** (which is an RNN) uses the encoder-decoder architecture. The encoder takes the input in one language, and the decoder outputs the translation to another language.



We can use **Autoencoder** (based on combining encoder-decoder architecture) to perform image segmentation (using convolutional layers).

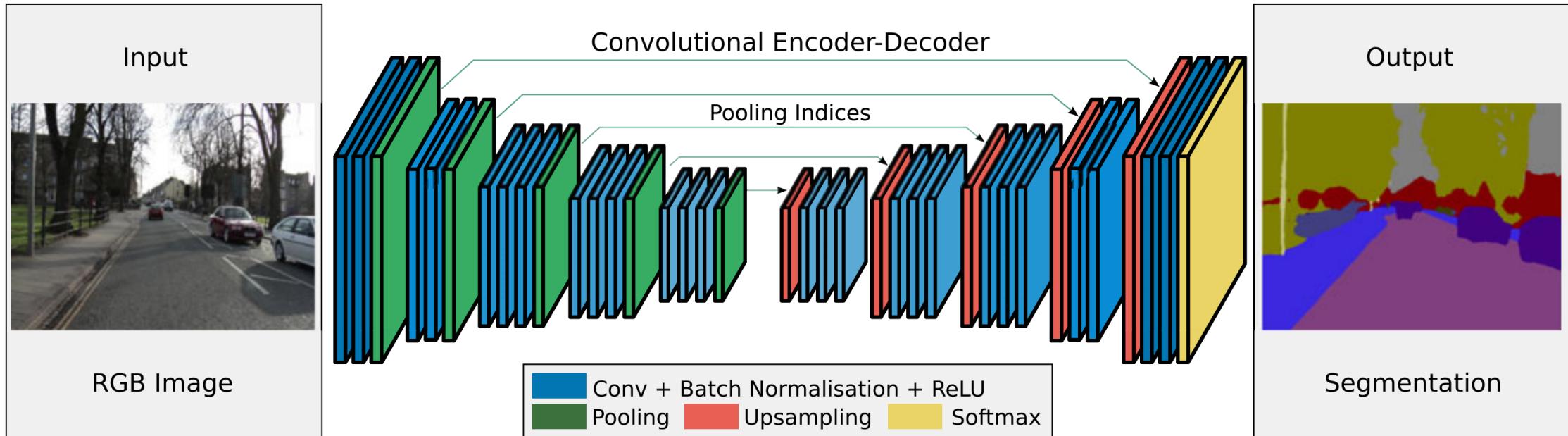


Fig. 2. An illustration of the SegNet architecture. There are no fully connected layers and hence it is only convolutional. A decoder upsamples its input using the transferred pool indices from its encoder to produce a sparse feature map(s). It then performs convolution with a trainable filter bank to densify the feature map. The final decoder output feature maps are fed to a soft-max classifier for pixel-wise classification.

A recent state-of-the-art is the **Transformer** architecture (based on encoder-decoder).

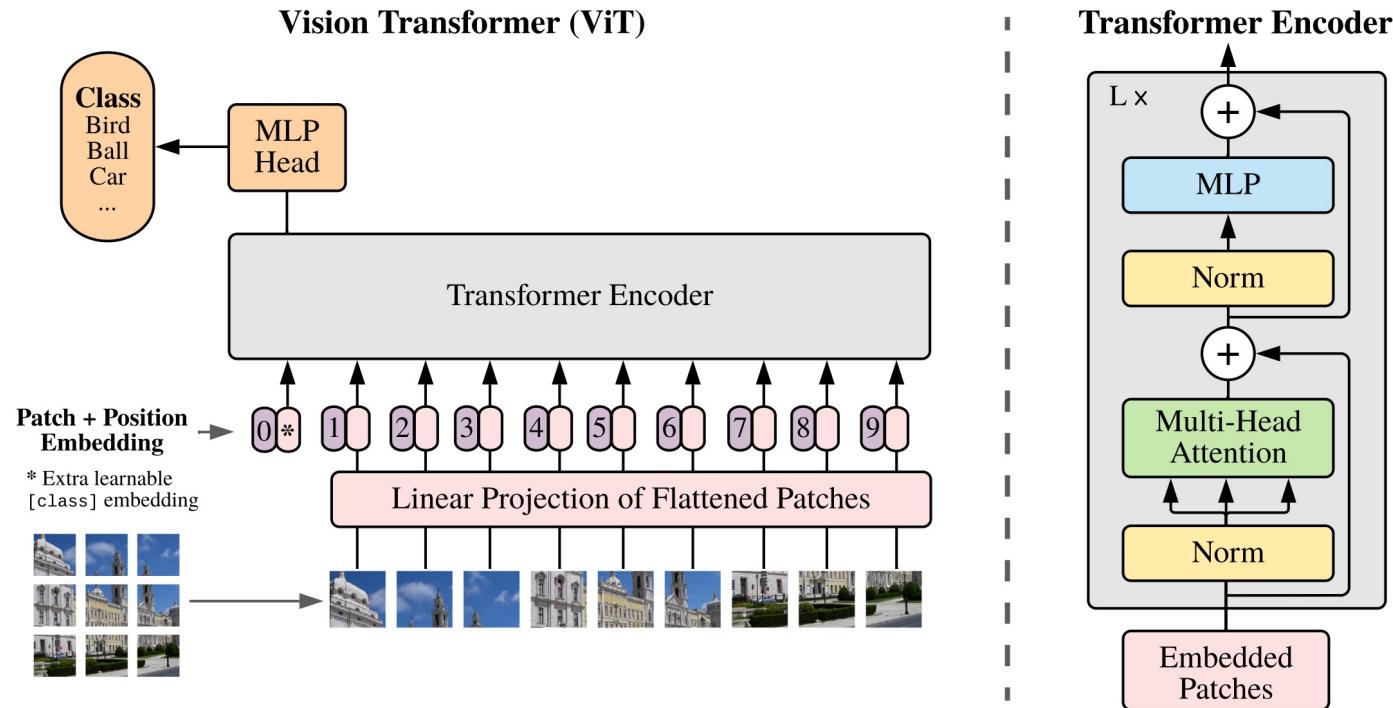
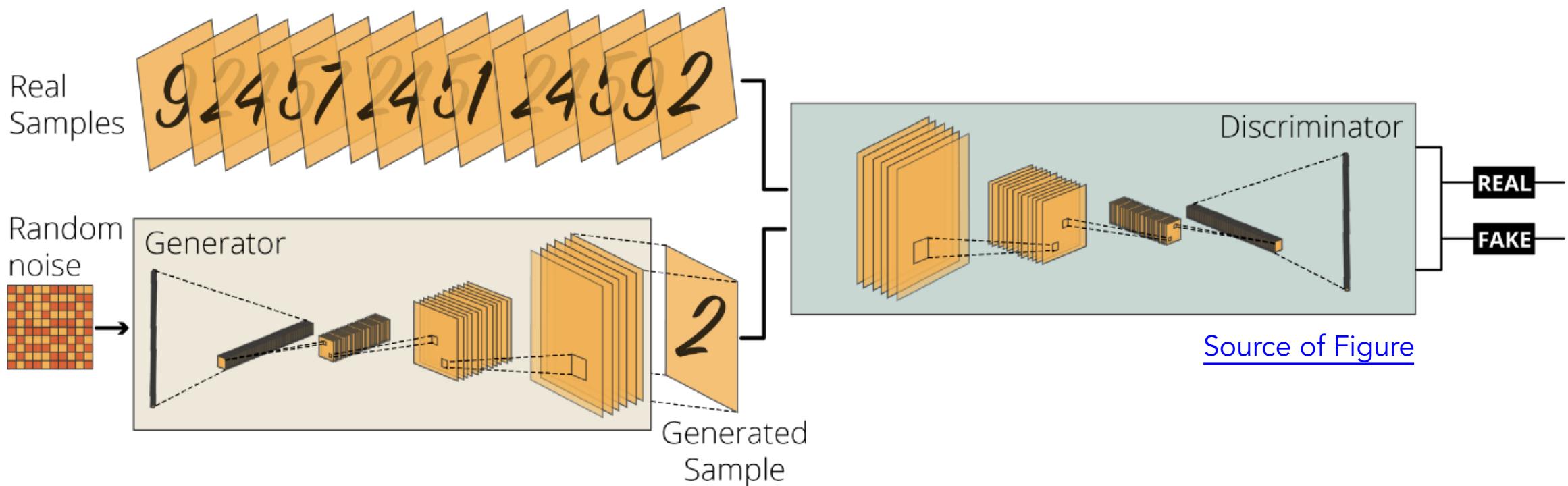
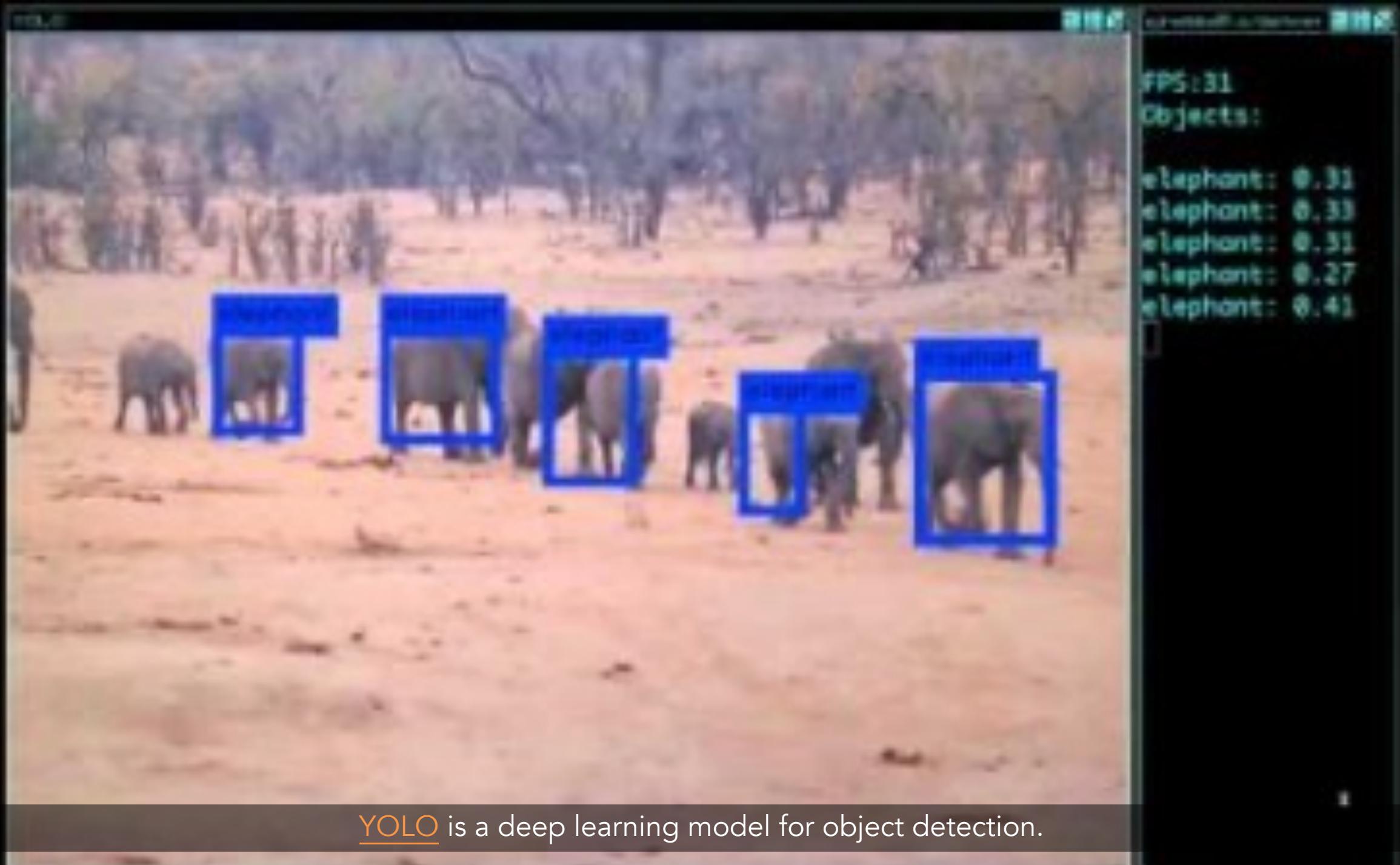


Figure 1: Model overview. We split an image into fixed-size patches, linearly embed each of them, add position embeddings, and feed the resulting sequence of vectors to a standard Transformer encoder. In order to perform classification, we use the standard approach of adding an extra learnable “classification token” to the sequence. The illustration of the Transformer encoder was inspired by Vaswani et al. (2017).

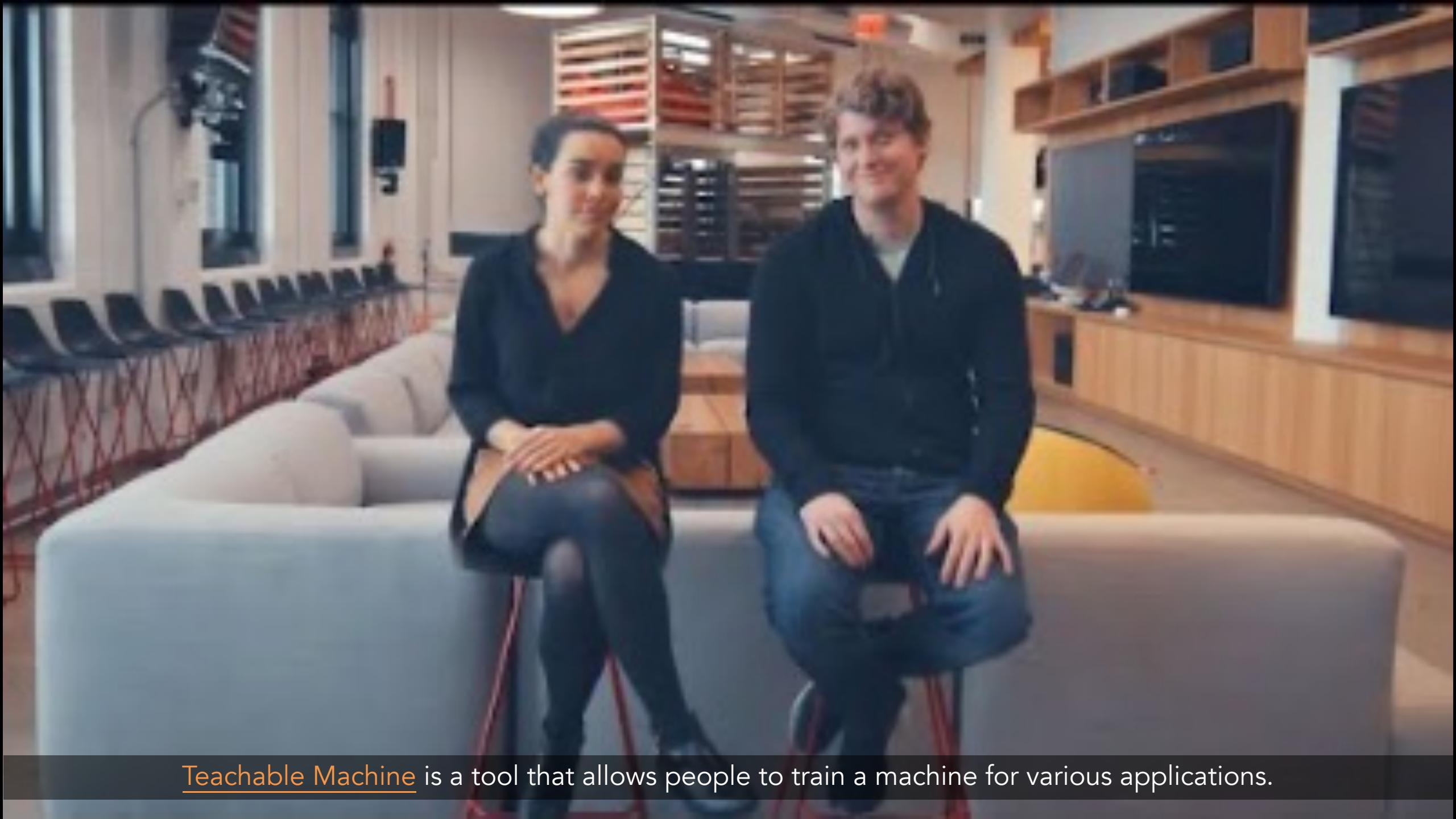
We can also use deep learning to generate data, such as the **Generative Adversarial Network** structure, which combines a generator (that converts noise to a fake sample) and a discriminator (that tries to identify if a sample is fake or real).



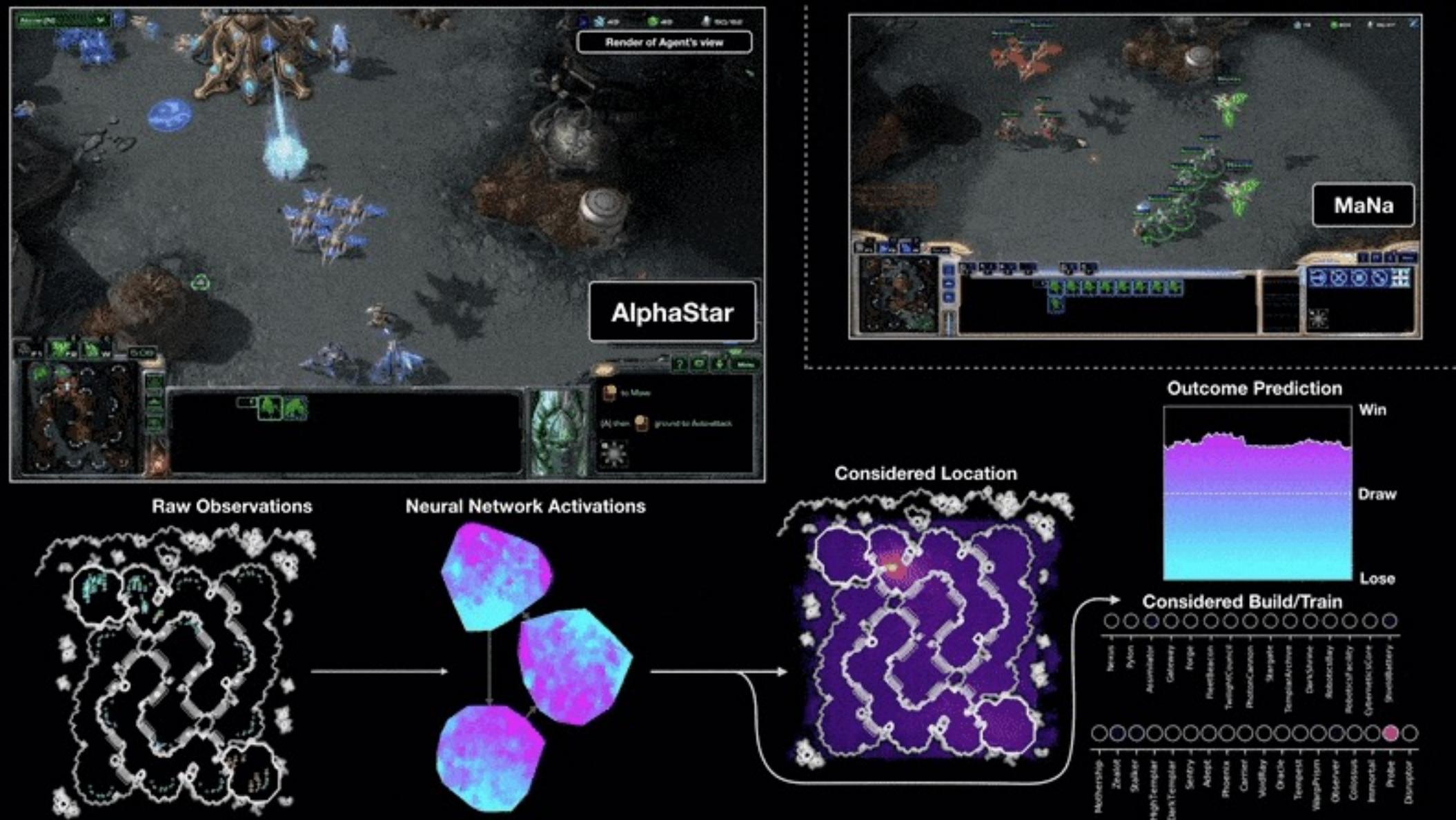




SegNet is a deep learning model for scene segmentation.



[Teachable Machine](#) is a tool that allows people to train a machine for various applications.



[This link](#) has a short highlight about how AlphaStar plays StarCraft II.



DALL-E 2 uses deep learning to create realistic images and art from text descriptions.

Stable Diffusion 2-1 - a HugginFace Model

huggingface.co/spaces/stabilityai/stable-diffusion

Spaces stabilityai/stable-diffusion like 9.77k Running on CPU UPGRADE

App Files Community 19457

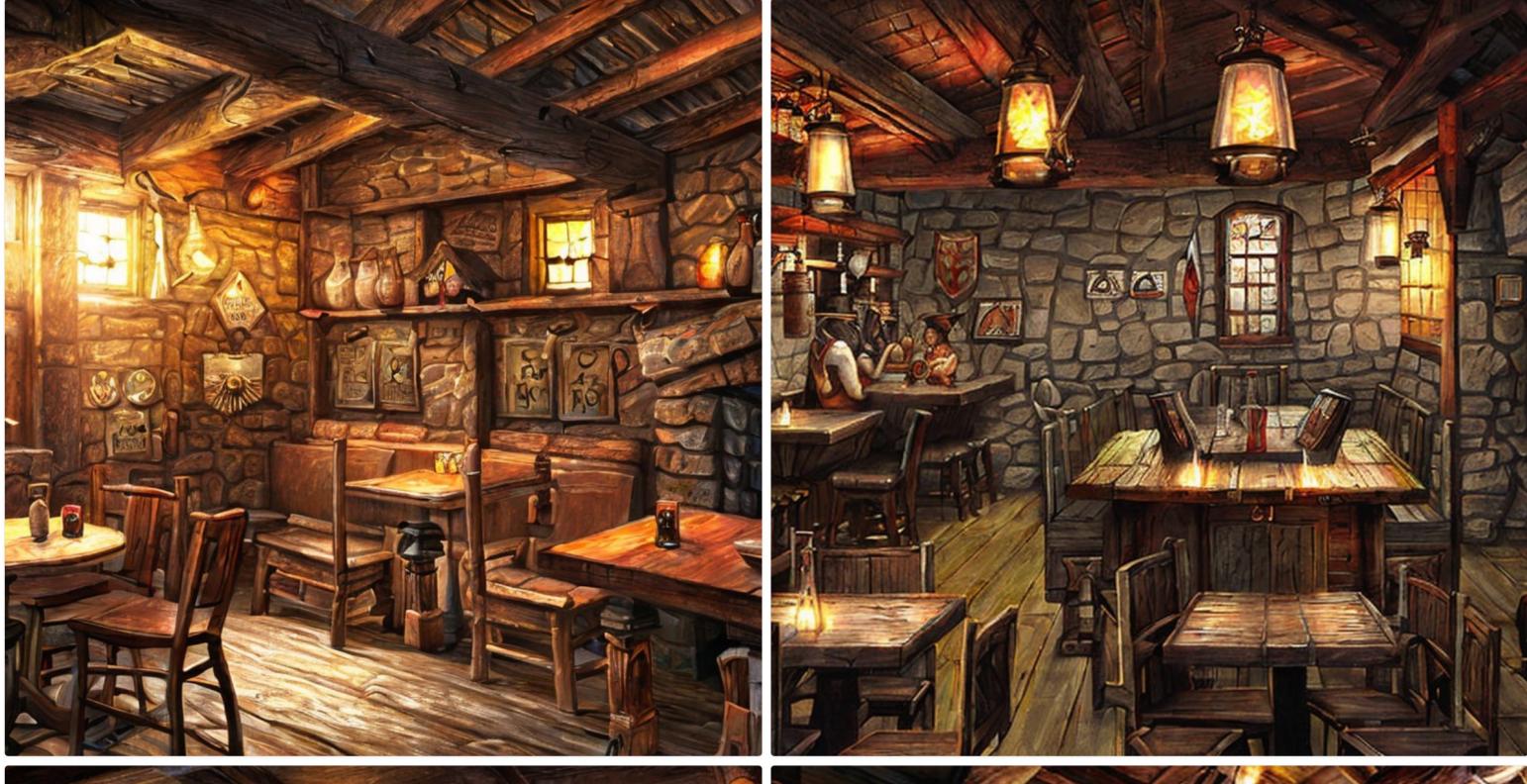
Stable Diffusion 2.1 Demo

Stable Diffusion 2.1 is the latest text-to-image model from StabilityAI. [Access Stable Diffusion 1 Space here](#)
For faster generation and API access you can try [DreamStudio Beta](#).

Interior of a tavern in a fantasy Dungeons & Dragons world in photorealistic style

low quality, ugly

Generate image



Stable Diffusion -- <https://huggingface.co/spaces/stabilityai/stable-diffusion>

Building Image Classifier with PyTorch

Explain the following python code:

```
def array_to_event(array, max_len=None):
    event = []
    array = copy.deepcopy(array)
    array.insert(0, 0) # insert a zero at the beginning
    if max_len is not None and max_len < 1: max_len = None
    for i in range(len(array)-1):
        a_i1 = array[i+1]
        diff = a_i1 - array[i]
        if diff == 1: # from 0 to 1
            event.append([i,i])
        if max_len == 1:
            array[i+1] = 0 # restart next event
        elif diff == 0: # from 0 to 0, or from 1 to 1
            if a_i1 == 1: # from 1 to 1
                event[-1][1] = i
            if max_len is not None and i-event[-1][0]+1 >= max_len:
                array[i+1] = 0 # restart next event
    return event
```

This Python function, `array_to_event`, takes an input array as its argument, and converts it into a list of events. An "event" here refers to a sequence of consecutive 1's in the input array.

The function first creates an empty list called `event`, which will store the events as they are detected. It then creates a copy of the input array using the `copy.deepcopy()` method, to avoid modifying the original array.

Next, the function inserts a 0 at the beginning of the copied array, to ensure that the first element in the input array is compared with 0.

ChatGPT Feb 13 Version. Free Research Preview. Our goal is to make AI systems more natural and safe to interact with. Your feedback will help us improve.

PyTorch CNN Training.

Can you provide an example about how to train a CNN model in pytorch?

Sure, here's an example of how to train a simple convolutional neural network (CNN) in PyTorch. In this example, we'll train a CNN to classify images from the CIFAR-10 dataset.

First, let's import the necessary packages:

```
python
Copy code
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
```

Next, we'll define the hyperparameters for our model:

```
python
Copy code
num_epochs = 10
batch_size = 64
learning_rate = 0.001
```

Now, we'll load the CIFAR-10 dataset and apply some data augmentation:

```
python
Copy code
transform_train = transforms.Compose[
```

ChatGPT Feb 13 Version. Free Research Preview. Our goal is to make AI systems more natural and safe to interact with. Your feedback will help us improve.

Building Image Classifier with PyTorch

Can you explain how gradient descent works?

Sure, I can explain how gradient descent works. Gradient descent is an optimization algorithm that is commonly used to minimize a function's value by iteratively adjusting its parameters.

Here's how it works:

1. First, you start with an initial set of parameters for the function you want to optimize.
2. Then, you calculate the gradient of the function with respect to each of the parameters. The gradient is a vector that points in the direction of steepest ascent of the function.
3. Next, you take a step in the opposite direction of the gradient to descend along the steepest slope of the function. The size of the step is determined by a hyperparameter called the learning rate.
4. You repeat this process, updating the parameters based on the negative gradient of the function, until you reach a minimum point or until you reach a stopping criterion.

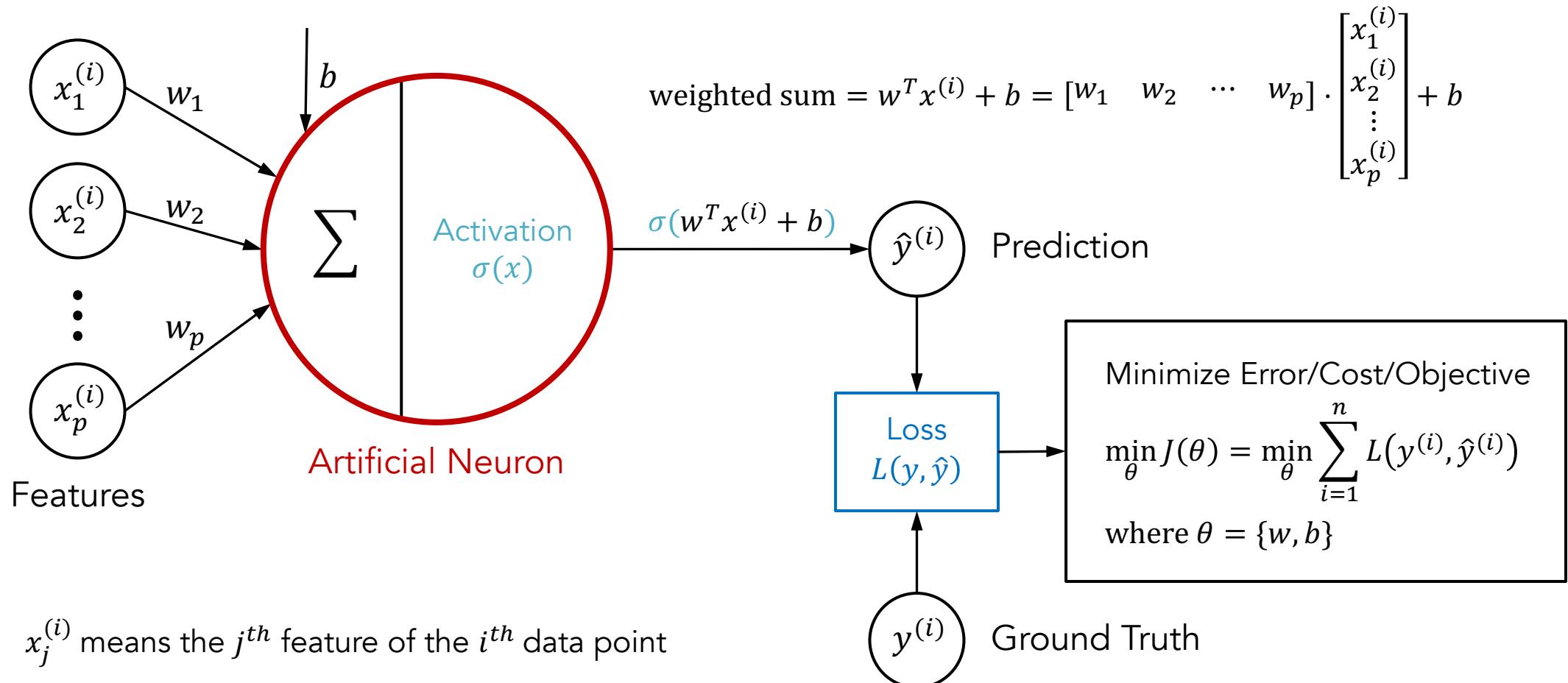
The goal of gradient descent is to find the set of parameters that minimize the value of the function. By following the negative gradient, we are iteratively moving towards the minimum point of the function.

There are different variations of gradient descent, including batch gradient descent, stochastic gradient descent, and mini-batch gradient descent, which differ in how they update the parameters and how they use the data to calculate the gradients.

ChatGPT Feb 13 Version. Free Research Preview. Our goal is to make AI systems more natural and safe to interact with. Your feedback will help us improve.

Now let us look at how a deep neural net
works, starting from the artificial neuron.

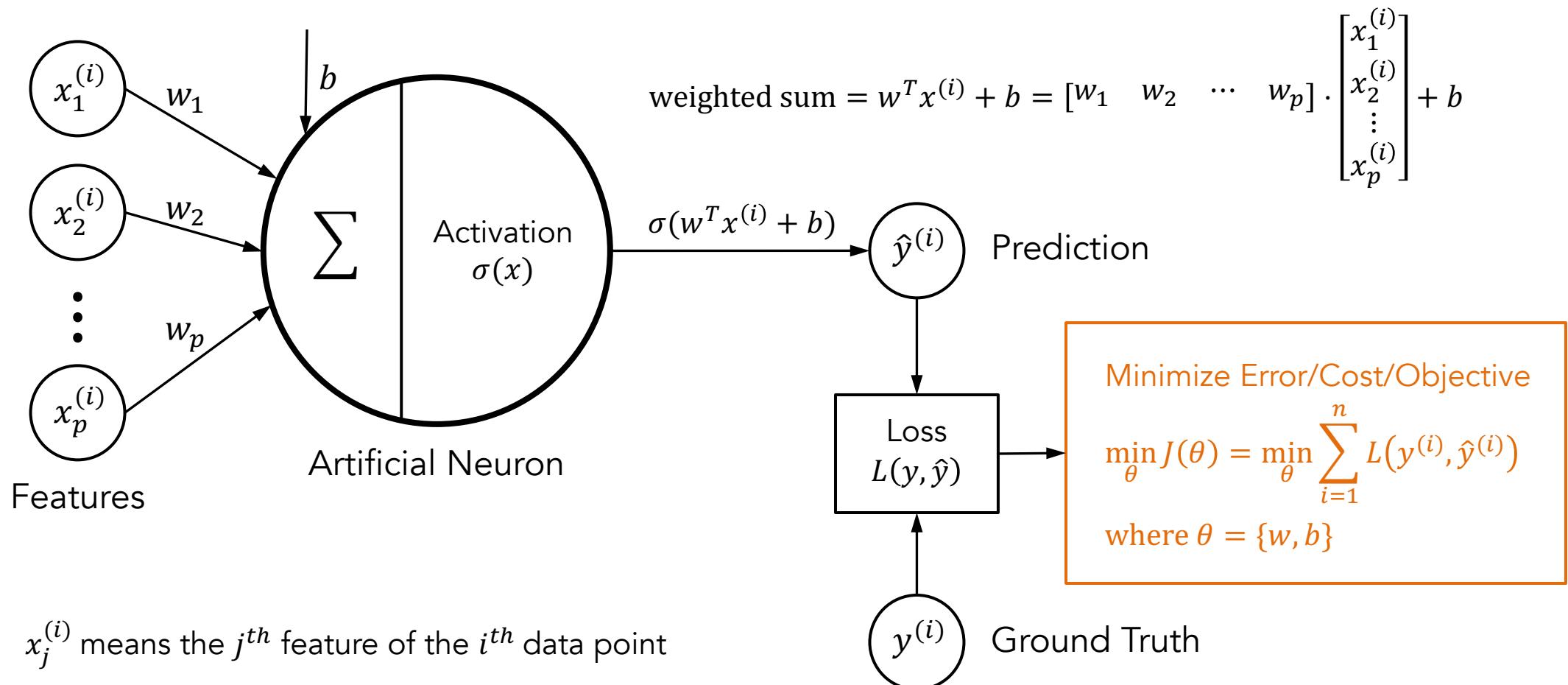
An **artificial neuron** can convert feature x to prediction \hat{y} by using a weighted sum and an **activation function**. Then, we need to define a **loss function** based on the task type.



$x_j^{(i)}$ means the j^{th} feature of the i^{th} data point

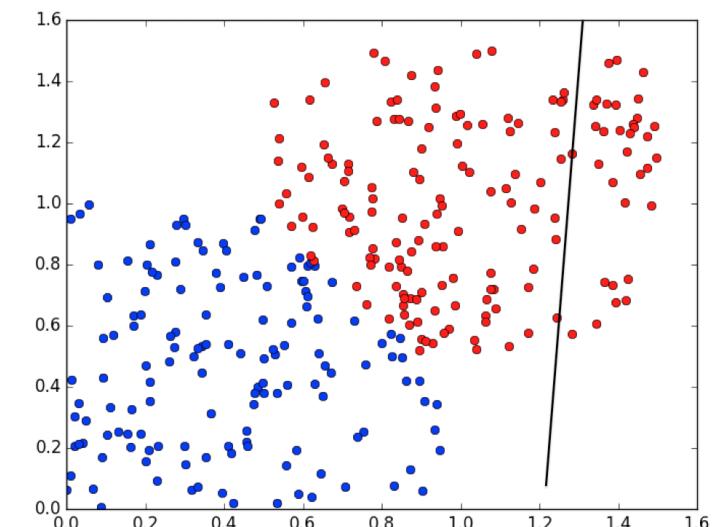
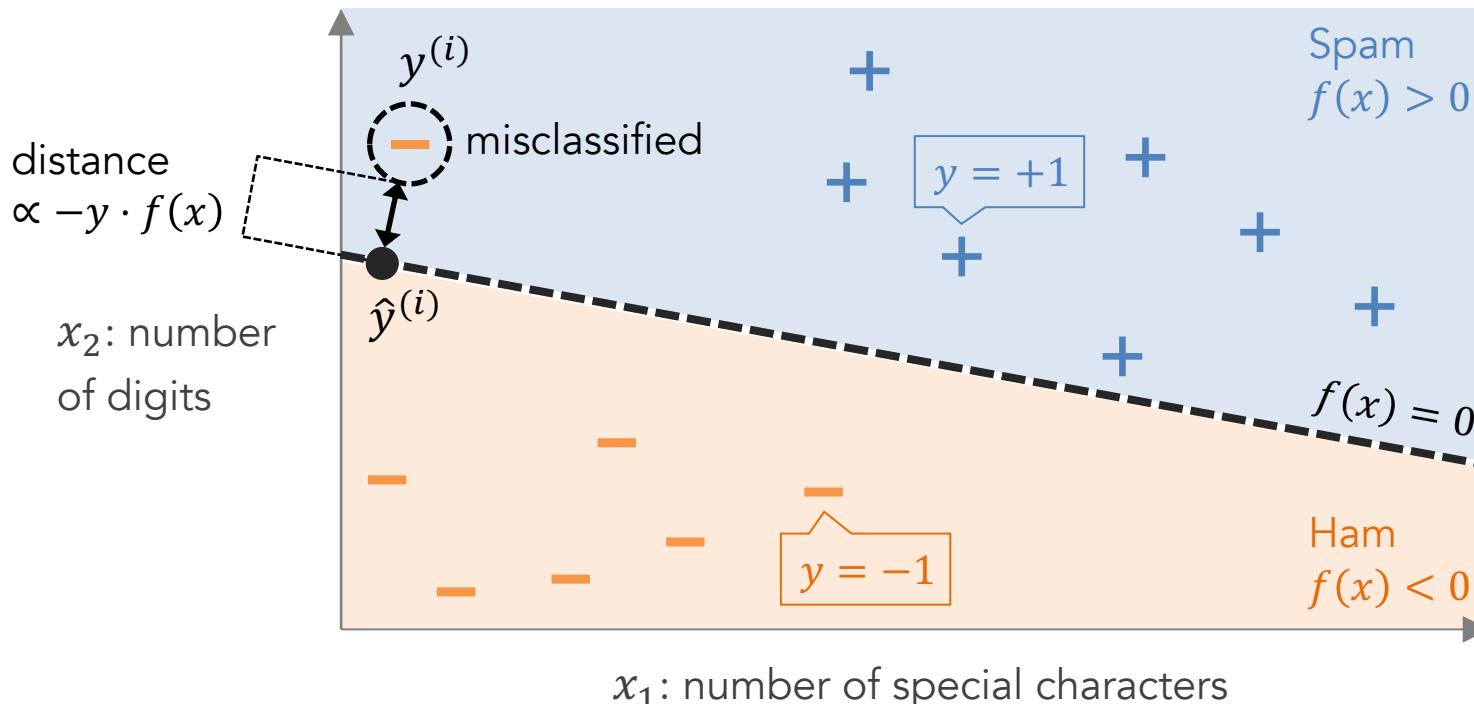
We need to **minimize the error/cost** to find the optimal parameter set $\theta = \{w, b\}$.

Notice that the prediction \hat{y} is a function of θ , meaning $\hat{y}(\theta) = \hat{y}(w, b) = \sigma(w^T x + b)$.

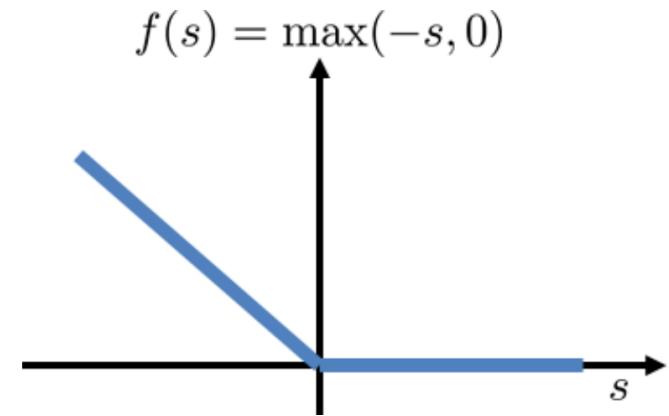
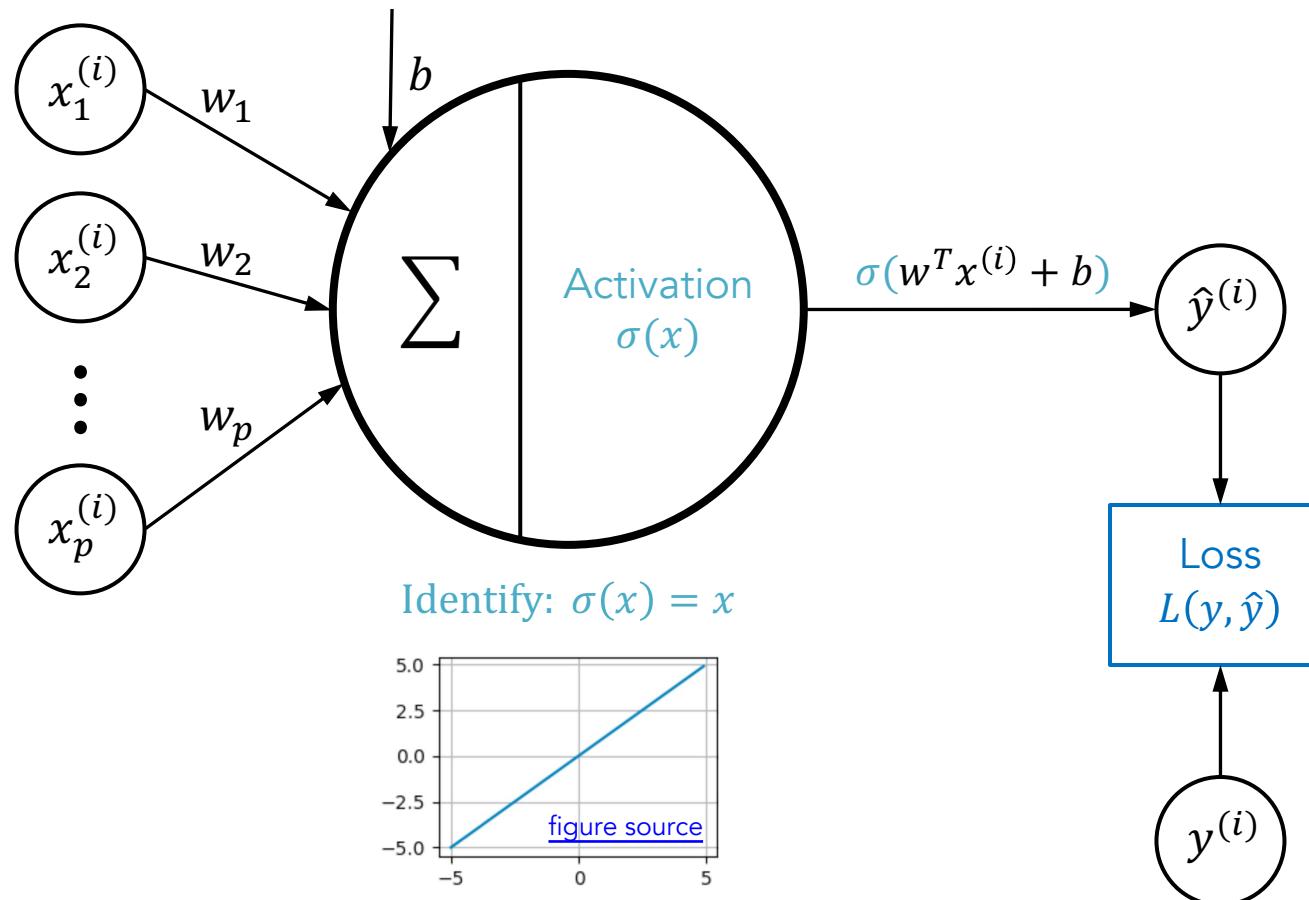


We have learned how to classify spam and ham using the perceptron algorithm with an error metric. We can also formulate the error in another way.

$$\text{error} = \sum_{i=1}^n -y^{(i)} \cdot f(x^{(i)}) \text{ for each misclassified point} = \sum_{i=1}^n \max(-y^{(i)} \cdot \hat{y}^{(i)}, 0) \text{ for all points}$$



We can represent the perceptron classifier using an artificial neuron. In this case, we use the **identity function (as the activation)** with the **soft perceptron loss**.

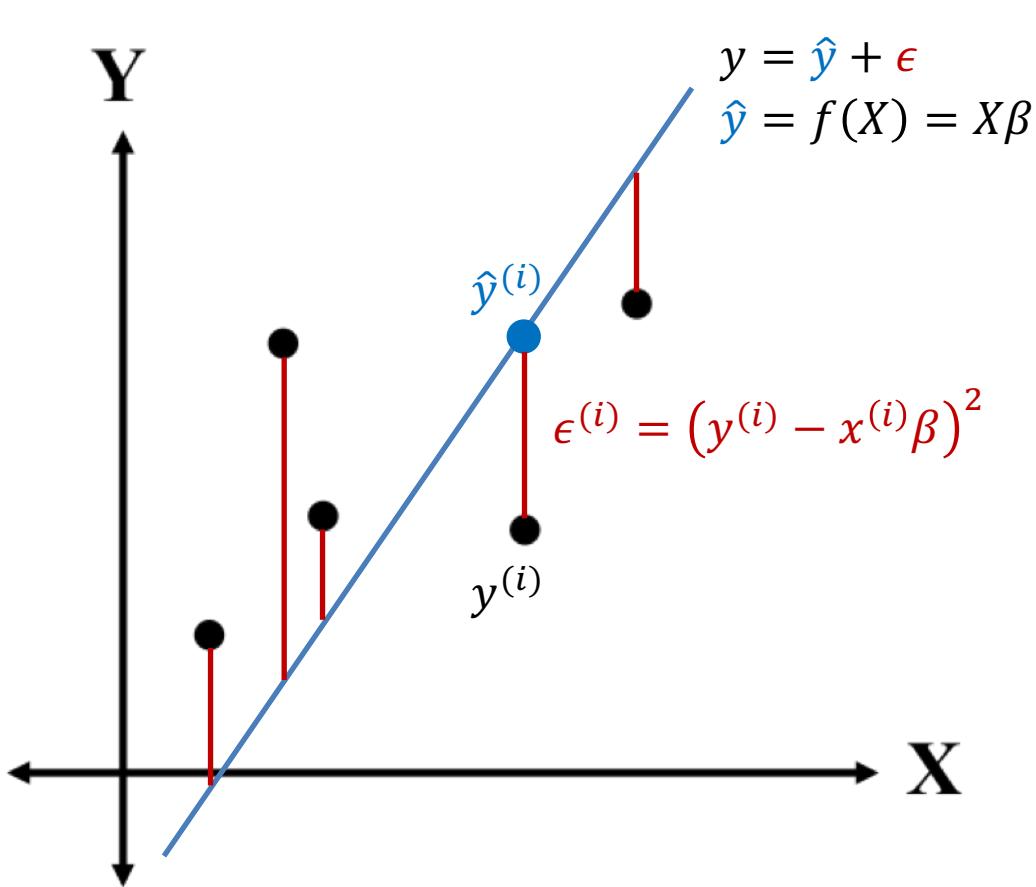


Soft perceptron loss

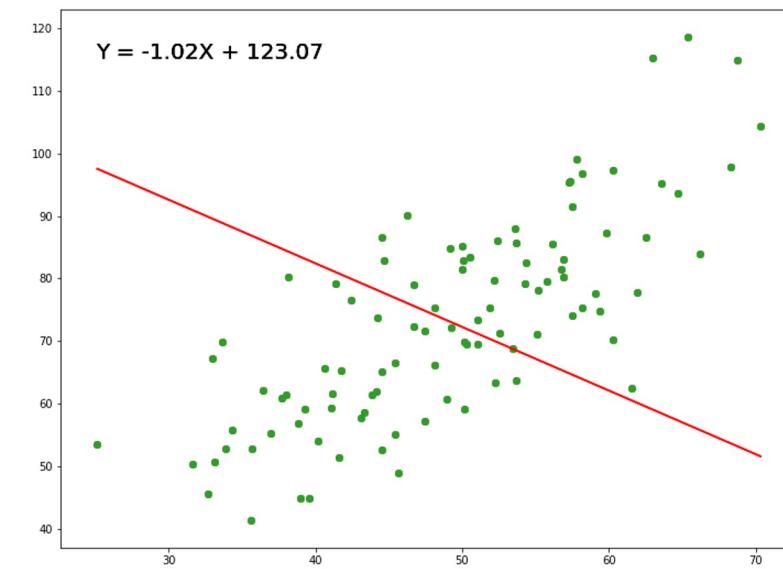
$$L(y, \hat{y}) = \max(-y \cdot \hat{y}, 0)$$

$$J(\theta) = \sum_{i=1}^n \max(-y^{(i)} \cdot \hat{y}^{(i)}, 0)$$

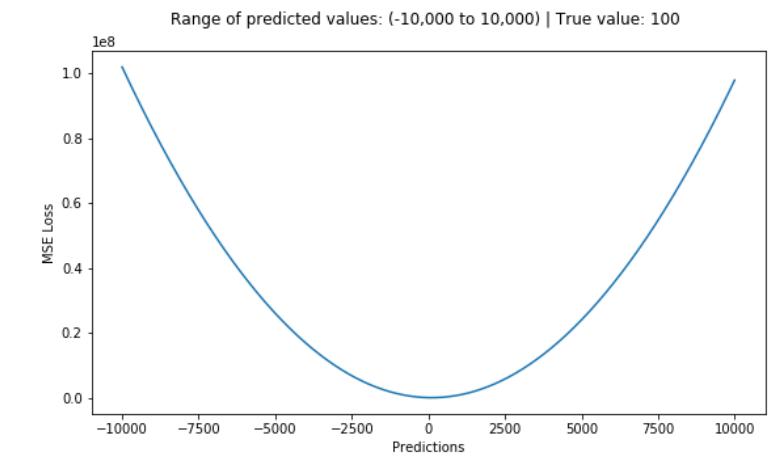
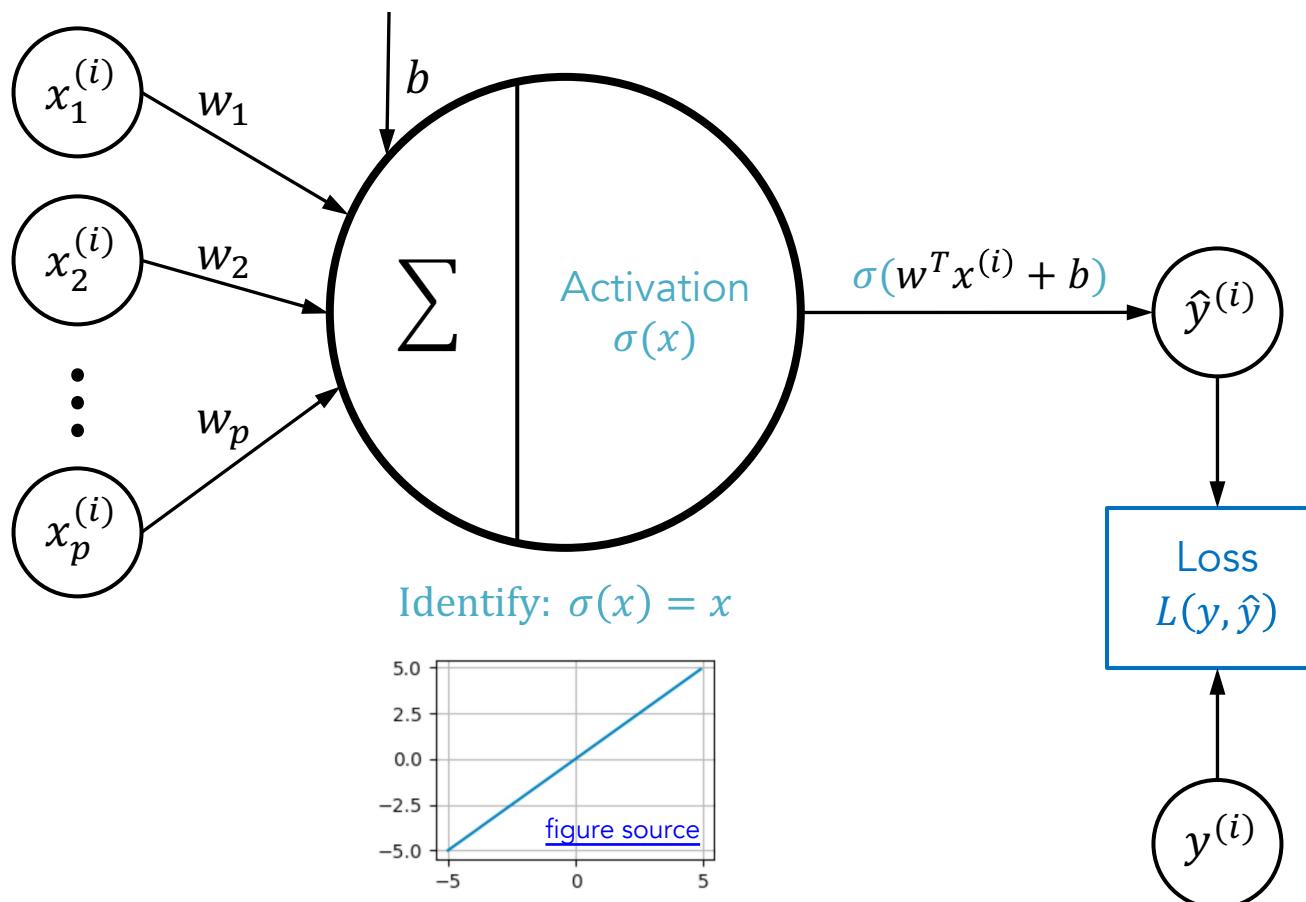
We have also learned linear regression, where we need to find a set of coefficients that can **minimize the squared distances** between all the predictions and ground truths.



$$\begin{aligned} \min_{\beta} \sum_{i=1}^n \epsilon^{(i)} &= \min_{\beta} \sum_{i=1}^n (y^{(i)} - x^{(i)}\beta)^2 \\ &= \min_{\beta} (y - X\beta)^T (y - X\beta) \end{aligned}$$



We can also represent the linear regression model using an artificial neuron. In this case, we use the **identity activation function** with the **squared error loss function**.

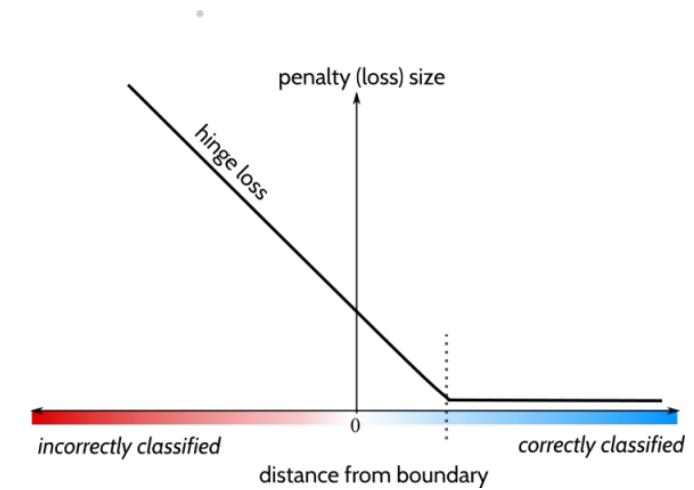
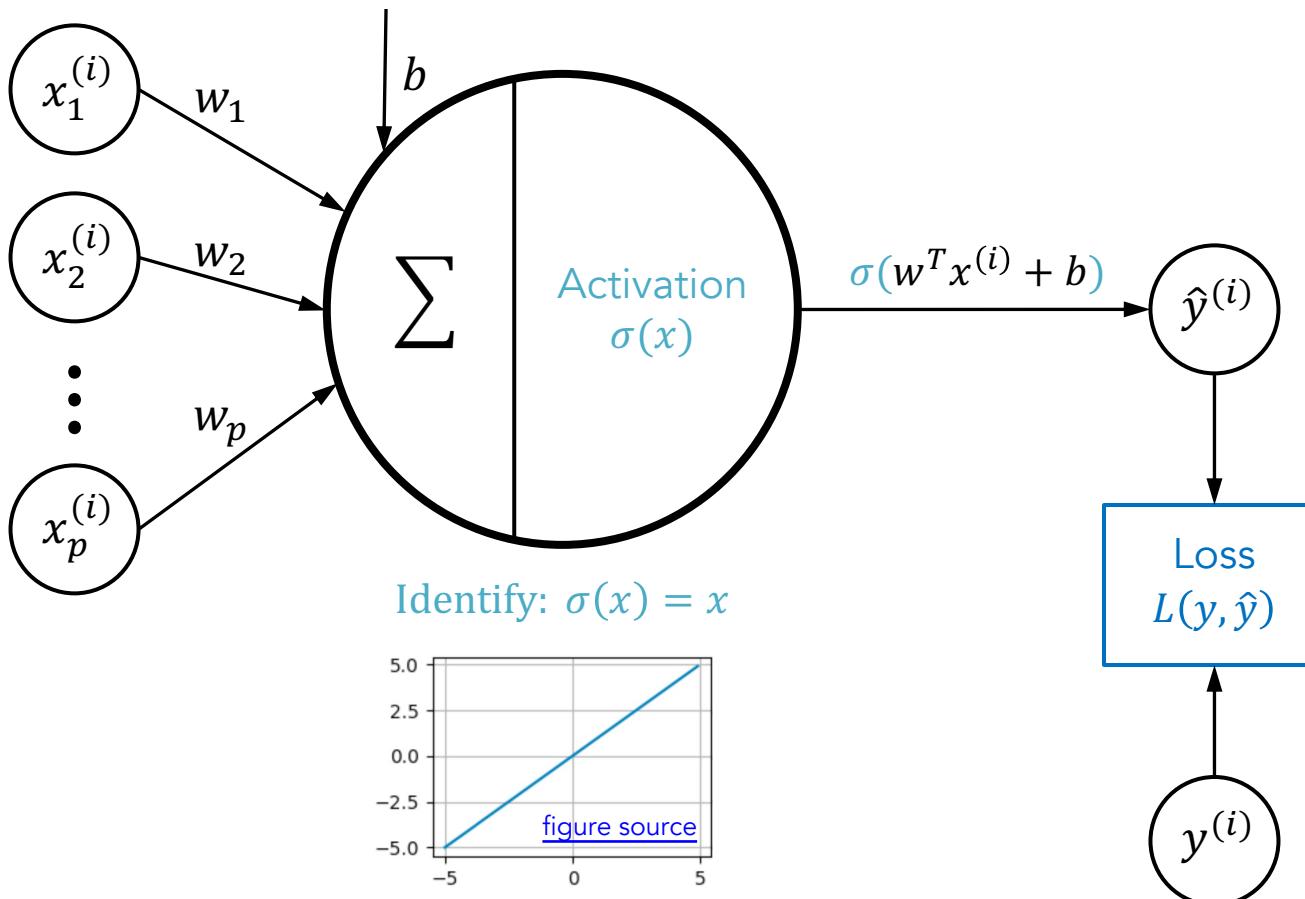


Squared error loss

$$L(y, \hat{y}) = (y - \hat{y})^2$$

$$J(\theta) = \sum_{i=1}^n (y^{(i)} - \hat{y}^{(i)})^2$$

We can replace the activation and loss functions with different ones to build another model. The example below uses hinge loss, which becomes **Support Vector Machine**.

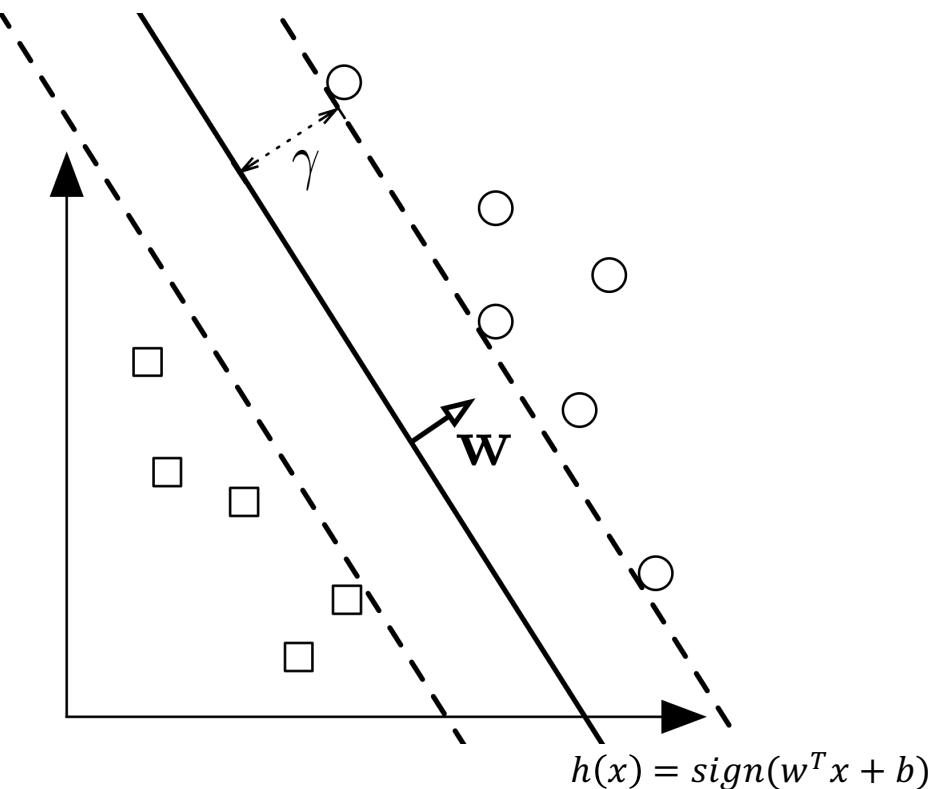
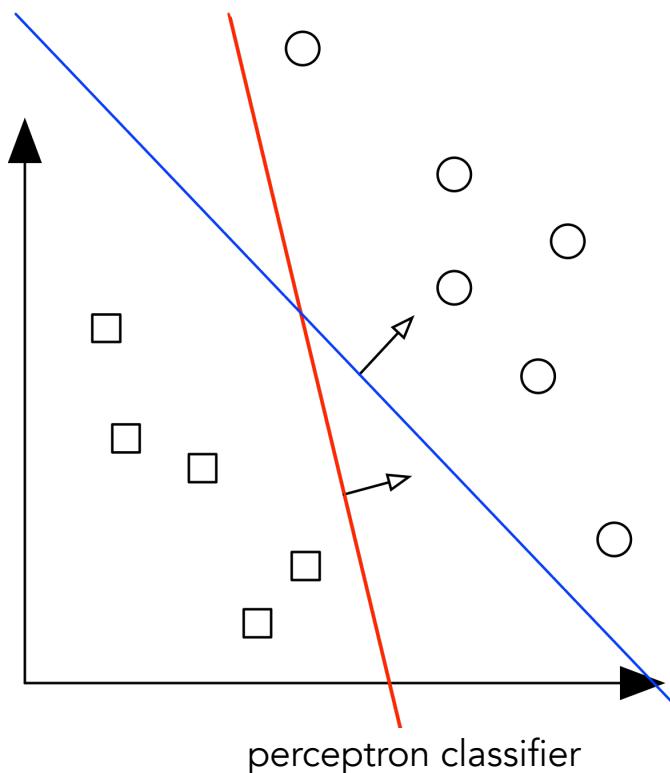


Hinge loss

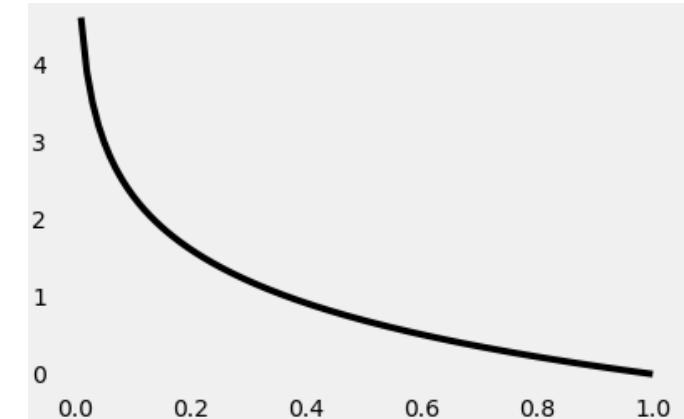
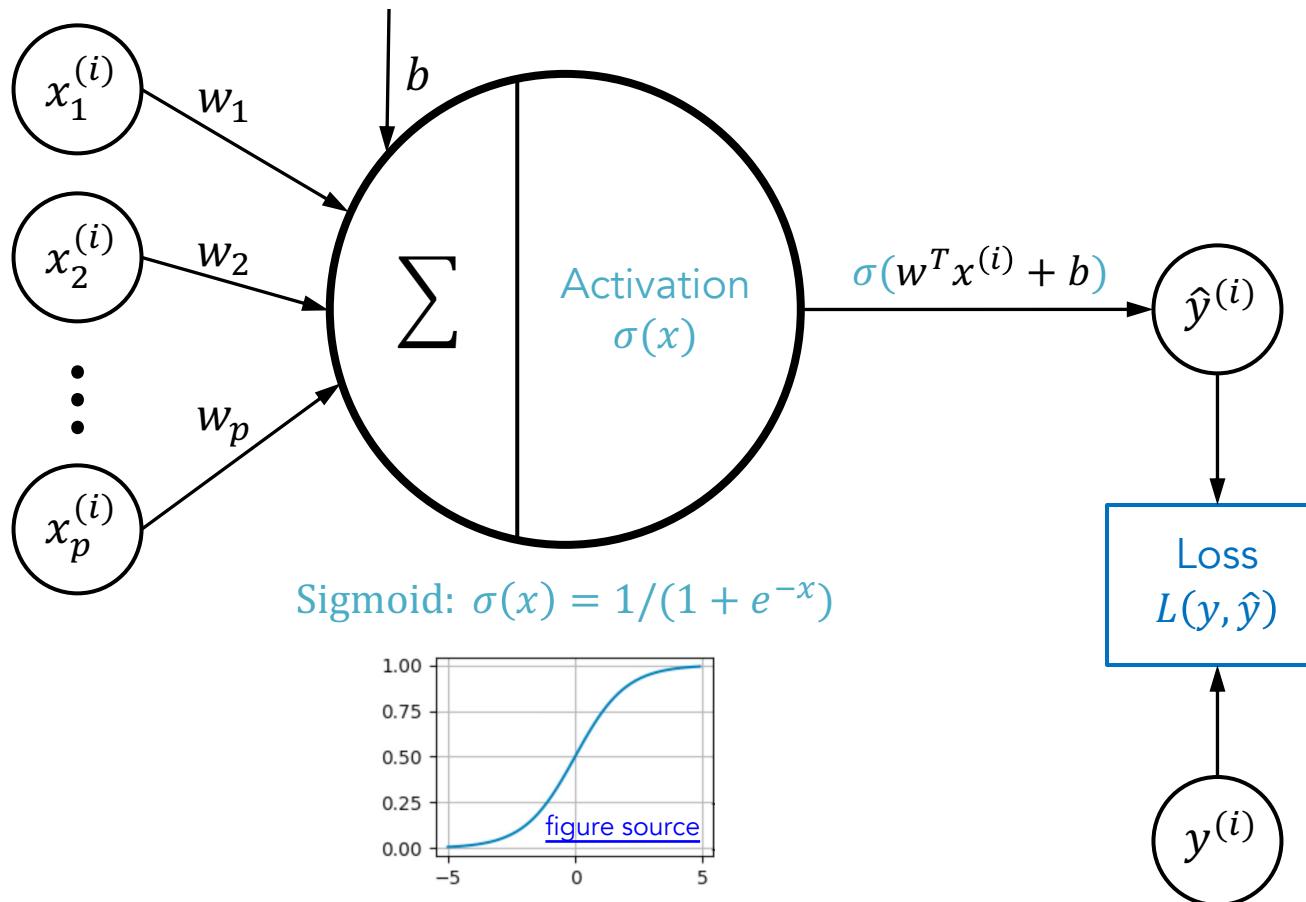
$$L(y, \hat{y}) = \max(1 - y \cdot \hat{y}, 0)$$

$$J(\theta) = \sum_{i=1}^n \max(1 - y^{(i)} \cdot \hat{y}^{(i)}, 0)$$

Support Vector Machine (right figure) finds the maximum margin γ separating hyperplane $h(x)$ for classification, while the perceptron classifier finds a separating hyperplane if it exists (e.g., blue or red line on the left figure).



If we replace the activation function to **sigmoid** and use the **logistic loss** (i.e., the binary version of cross-entropy loss), the neuron becomes a **Logistic Regression** model.

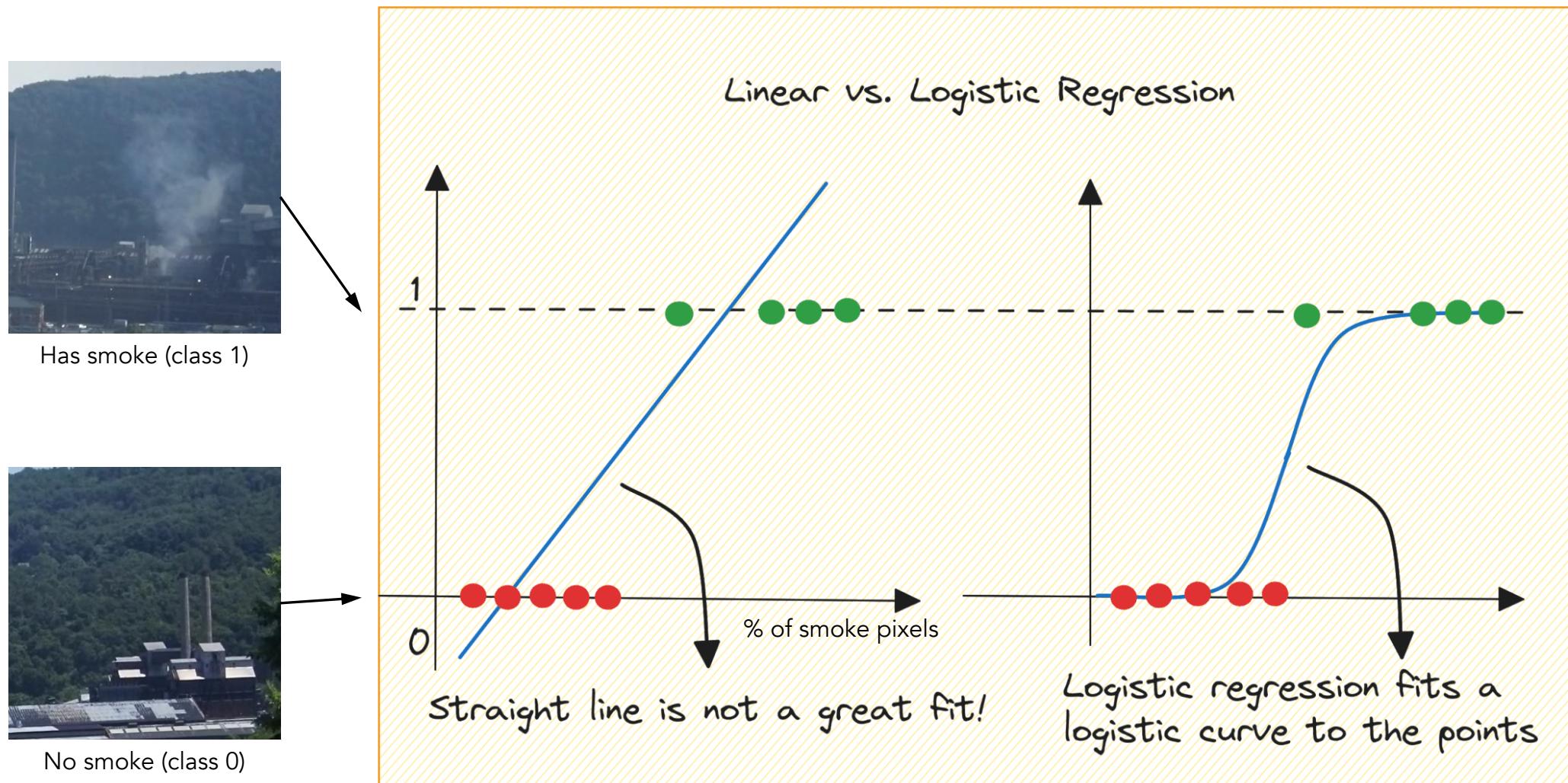


Binary cross-entropy loss

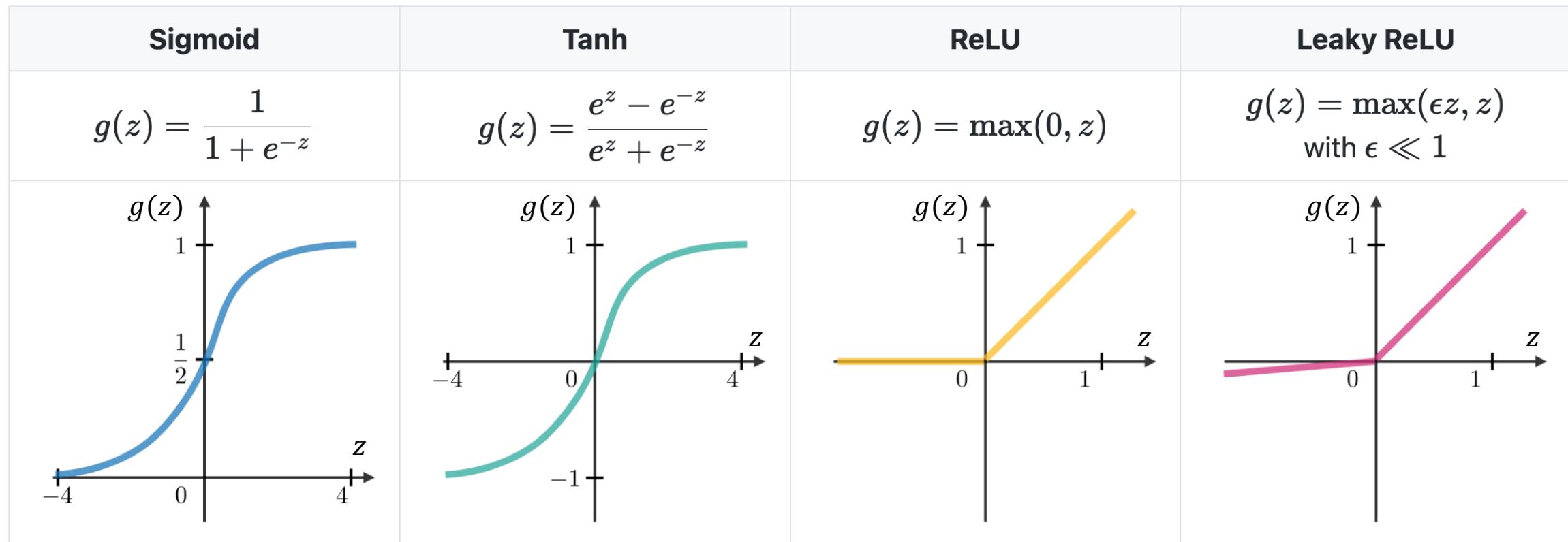
$$L(y, \hat{y}) = - \sum_{c=1}^2 y_c \log \hat{y}_c \text{ for 2 classes}$$

$$J(\theta) = - \sum_{i=1}^n \sum_{c=1}^2 y_c^{(i)} \log \hat{y}_c^{(i)}$$

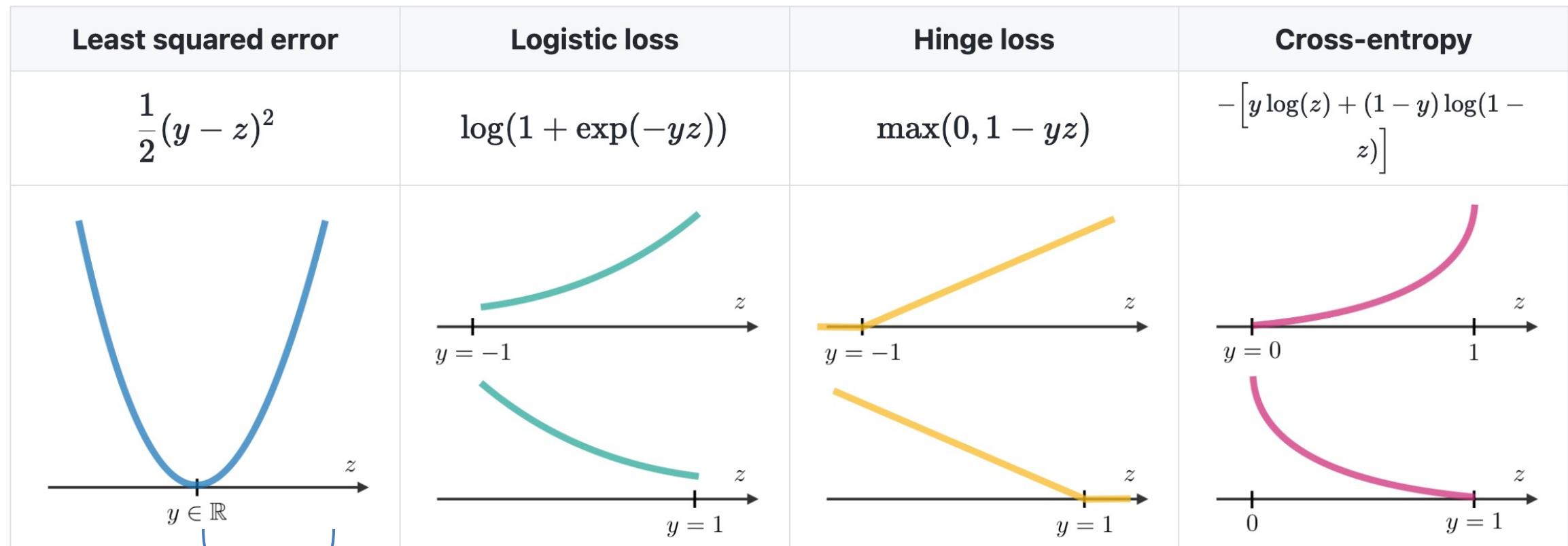
Logistic Regression model fits a logistic curve to the data to perform classification tasks.



Many activation functions exist for various purposes. For example, in classification, ReLU is typical for the middle layers, and sigmoid (or softmax) is for the final layer.



Also, there are different loss functions for different types of tasks. For example, the least squared error is for regression, and the others below are for classification.



How different is z from y , where y is the ground truth, and z is the prediction.

The major reason of using activation functions is to introduce non-linearity.

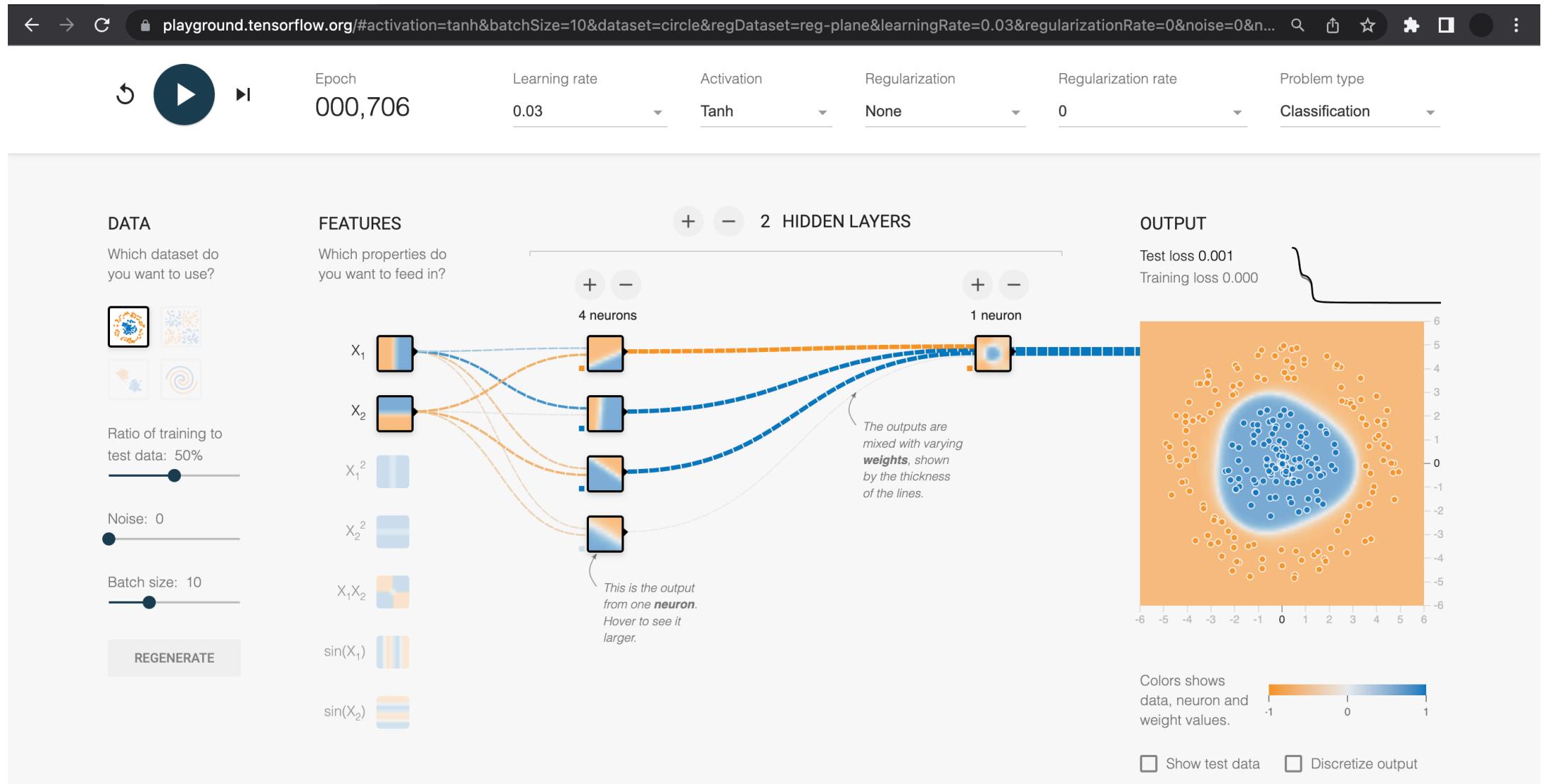


Figure source -- [the neural network playground using the Tanh activation function](https://playground.tensorflow.org/#activation=tanh&batchSize=10&dataset=circle®Dataset=reg-plane&learningRate=0.03®ularizationRate=0&noise=0&nHidden=2&nNeurons=4&w0=-0.7&b0=0.1&w1=-0.3&b1=0.1&w2=0.9&b2=-0.7&showData=1&discretizeOutput=1)

For example, the identity activation function does not help in grouping non-linear data.

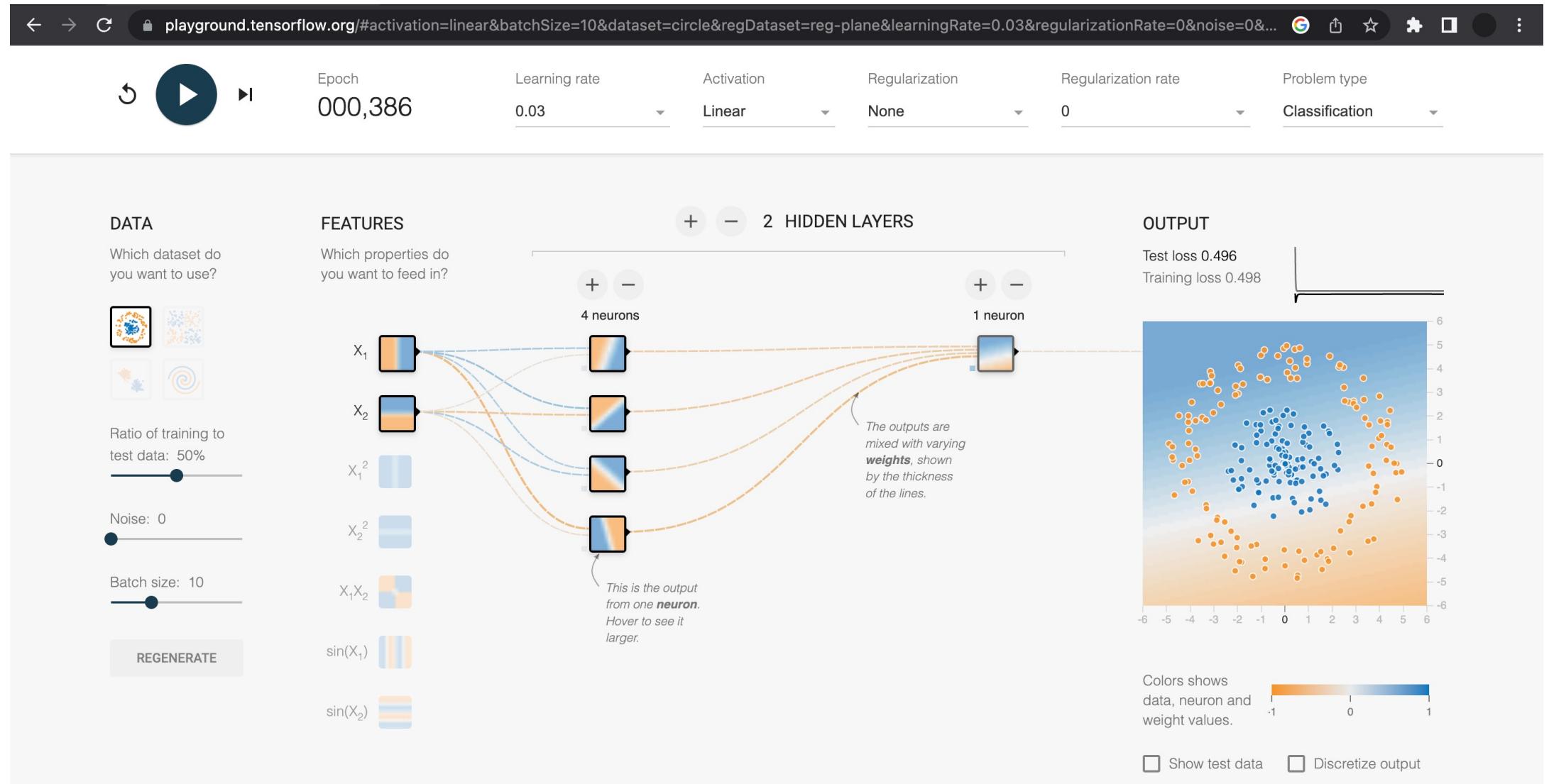
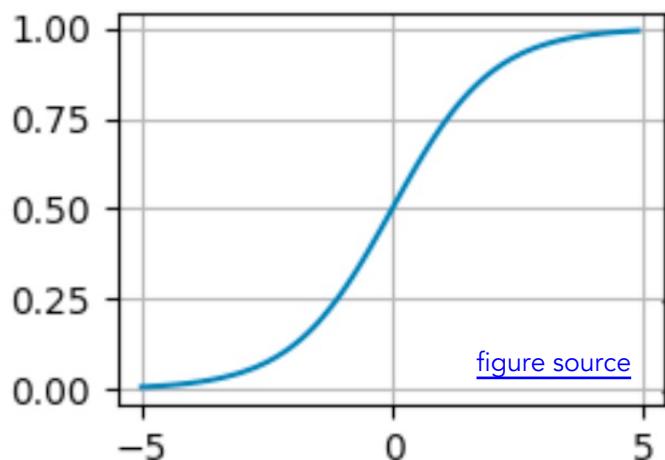


Figure source -- [the neural network playground using the identity activation function](https://playground.tensorflow.org/#activation=linear&batchSize=10&dataset=circle®Dataset=reg-plane&learningRate=0.03®ularizationRate=0&noise=0&...)

Why do we use a sigmoid function (or similar) as the activation function σ for the classification task?

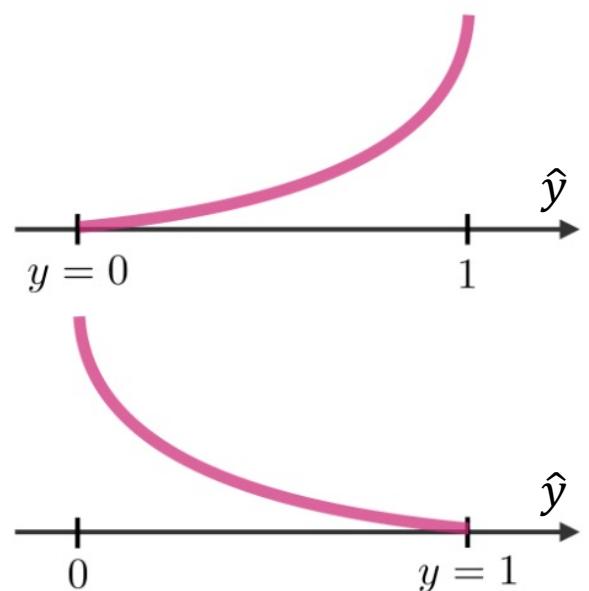
Sigmoid: $\sigma(x) = 1/(1 + e^{-x})$



Hint: think about the following:

- $\hat{y} = \sigma(w^T x + b)$
- What is the role of σ ?
- What does the predicted output \hat{y} look like?

Why do we use cross entropy as the loss function for the classification task?



Hint: think about the difference between entropy and cross entropy:

$$\text{Entropy} = \sum_{\text{Class } c} y_c \cdot \log_2 \left(\frac{1}{y_c} \right)$$

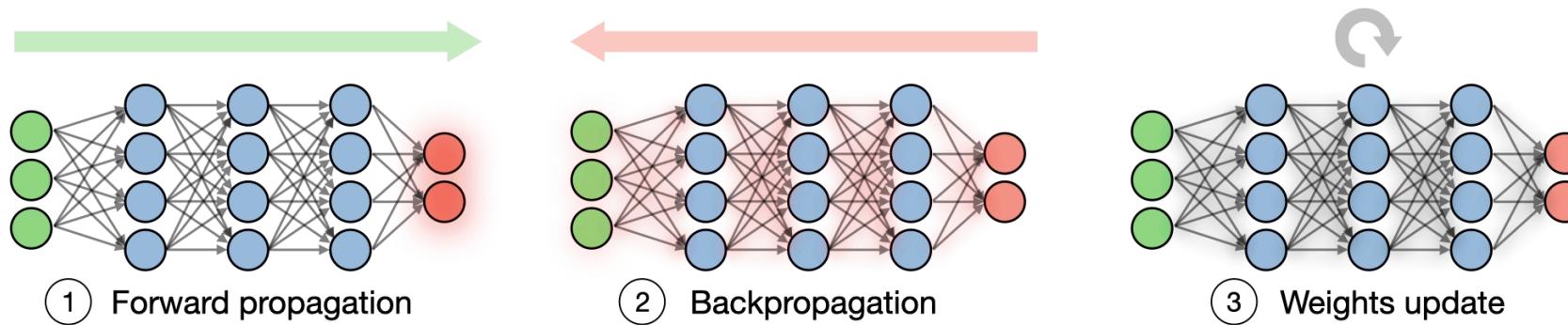
$$\text{Cross Entropy} = \sum_{\text{Class } c} y_c \cdot \log_2 \left(\frac{1}{\hat{y}_c} \right)$$

We can think about cross entropy loss as the exam, the true value y_c as the **teacher** (from the training data), and the predicted value \hat{y}_c as the **student** (from the model).

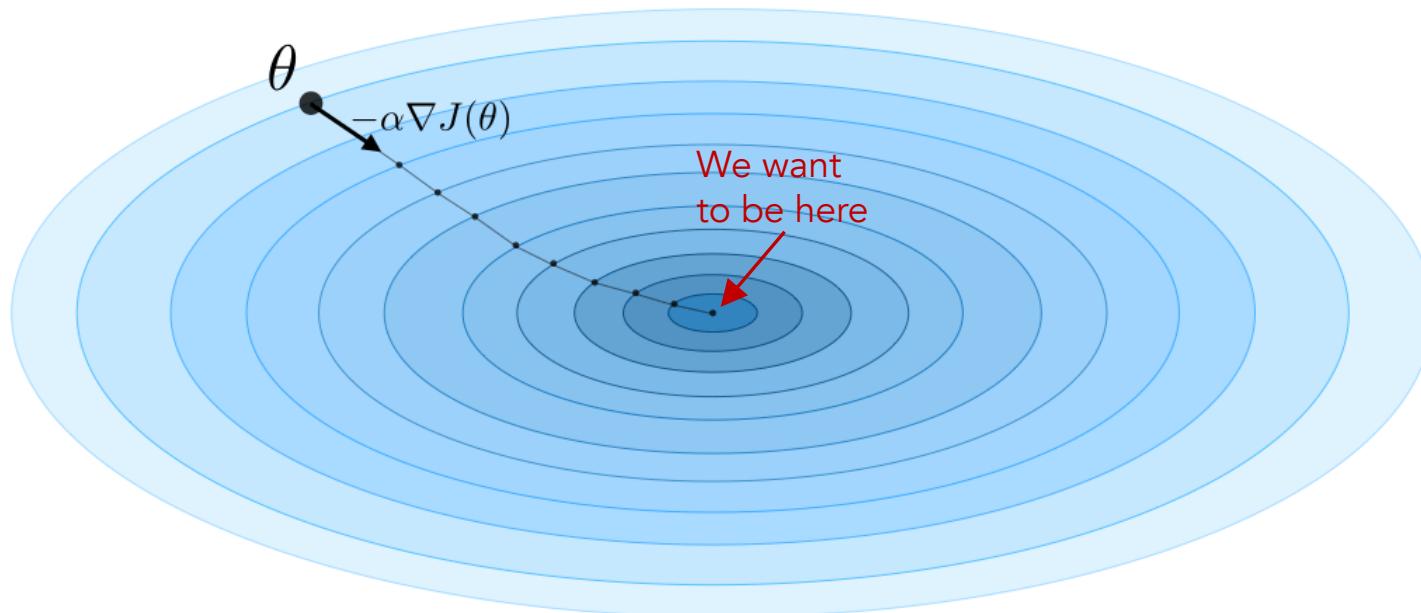
$$\text{Cross Entropy} = \sum_{\text{Class } c} y_c \cdot \log_2\left(\frac{1}{\hat{y}_c}\right)$$

The diagram illustrates the components of cross entropy loss. It features three black dots representing data points. The first dot is labeled 'Exam' with a black arrow pointing to it. The second dot is labeled 'Teacher' with a blue arrow pointing to it. The third dot is labeled 'Student' with a red arrow pointing to it. Dotted lines connect the 'Exam' dot to the 'Teacher' dot and the 'Teacher' dot to the 'Student' dot, representing the flow of information or loss calculation.

How can we train the deep neural network?



We need to use optimization algorithms, such as **gradient descent**, to help us find a local minimum (or global minimum for convex functions) on the **cost function**. We need to set a **learning rate**, which means the pace of moving forward for each update.



$$\min_{\theta} J(\theta) = \min_{\theta} \sum_{i=1}^n L(y^{(i)}, \hat{y}^{(i)})$$

$$\theta_{t+1} \leftarrow \theta_t - \alpha \cdot \nabla J(\theta_t)$$

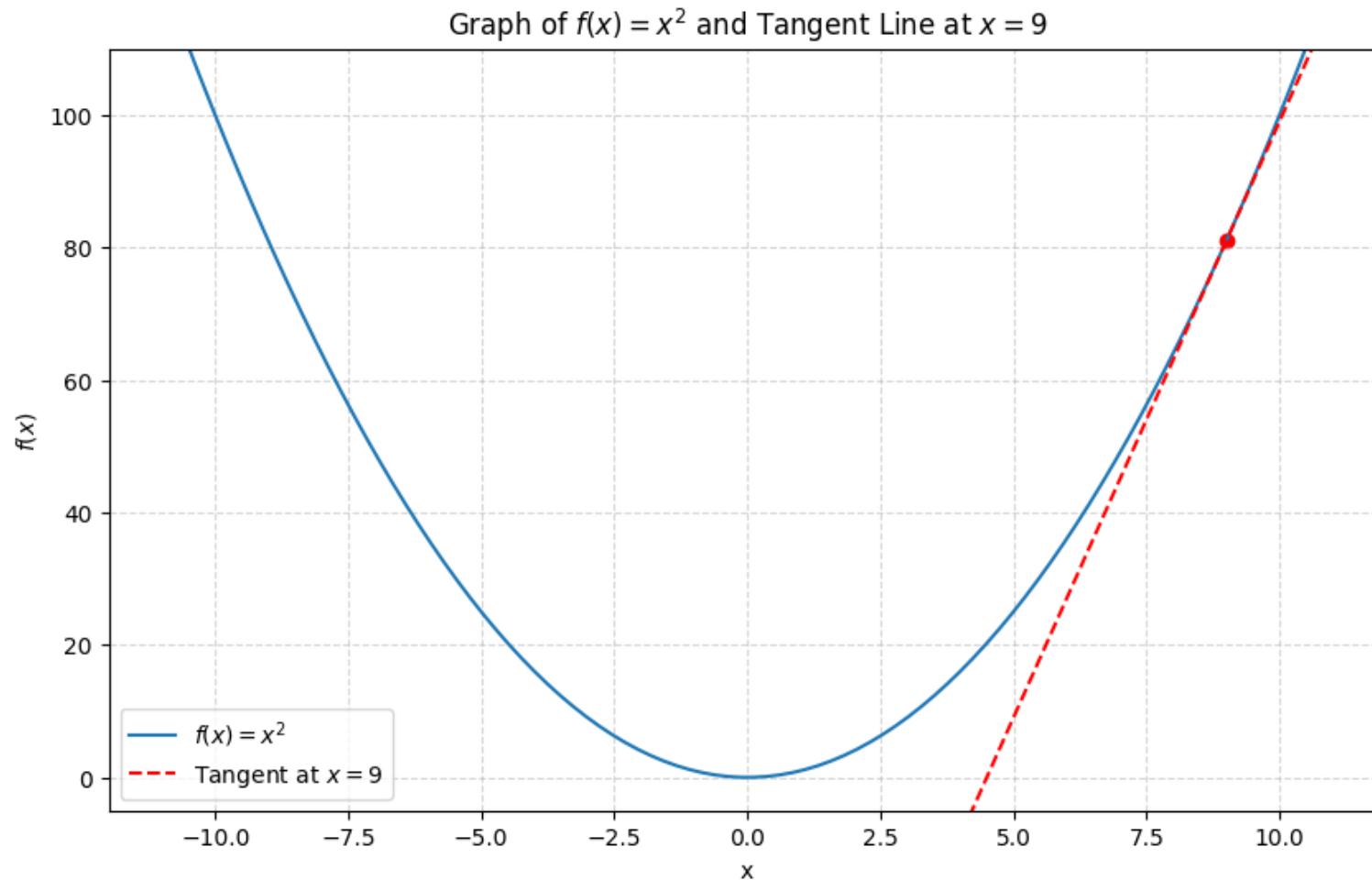
θ : model parameters

α : learning rate

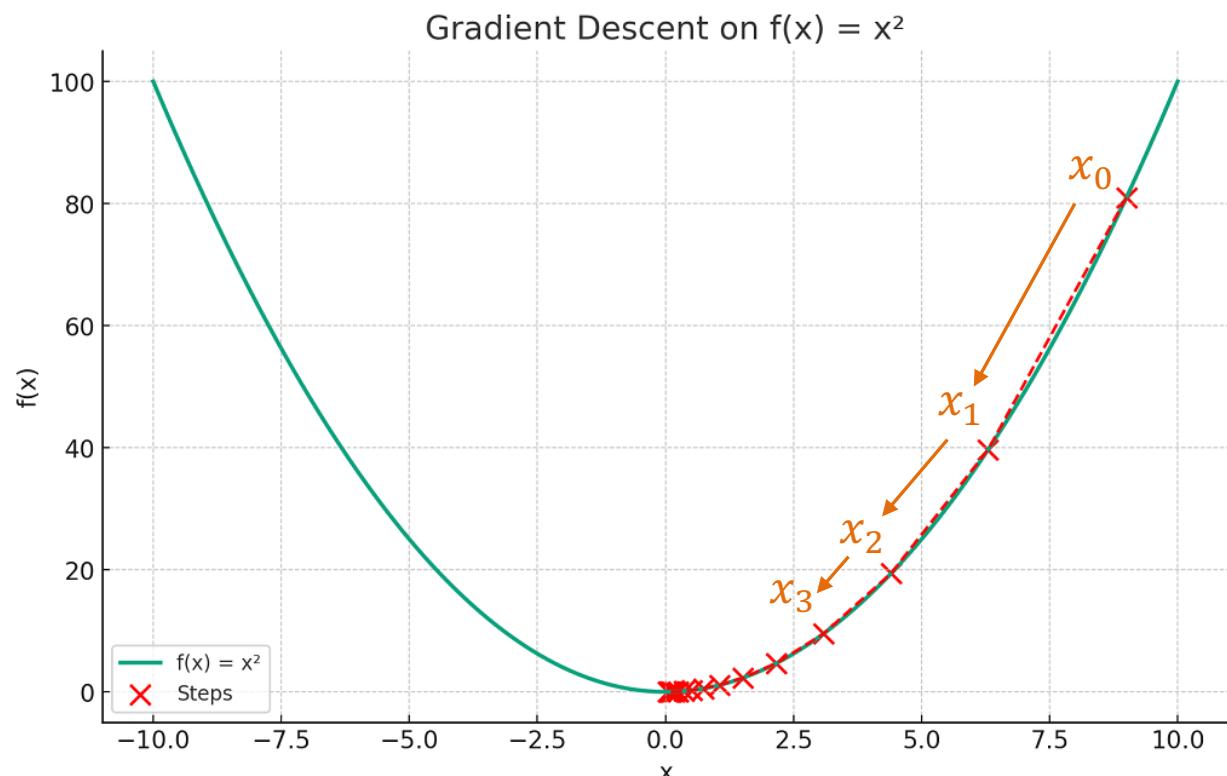
t : training step

$\nabla J(\theta)$: gradient

Gradient is a **generalization of derivative**. The intuition is that computing the derivative of $f(x)$ at point x_t means computing the slope of the tangent line to $f(x)$ at point x_t .



Exercise 5.1: Perform three gradient descent updates (i.e., compute x_1 , x_2 , and x_3) for one parameter x with the cost function $f(x) = x^2$, using the starting point $x_0 = 9$ and learning rate $\alpha = 0.1$ (with the provided gradient function $\nabla f(x)$ below).



$$f(x) = x^2$$

$$\nabla f(x) = 2x$$

$$x_{t+1} \leftarrow x_t - \alpha \cdot \nabla f(x_t)$$

x : model parameter

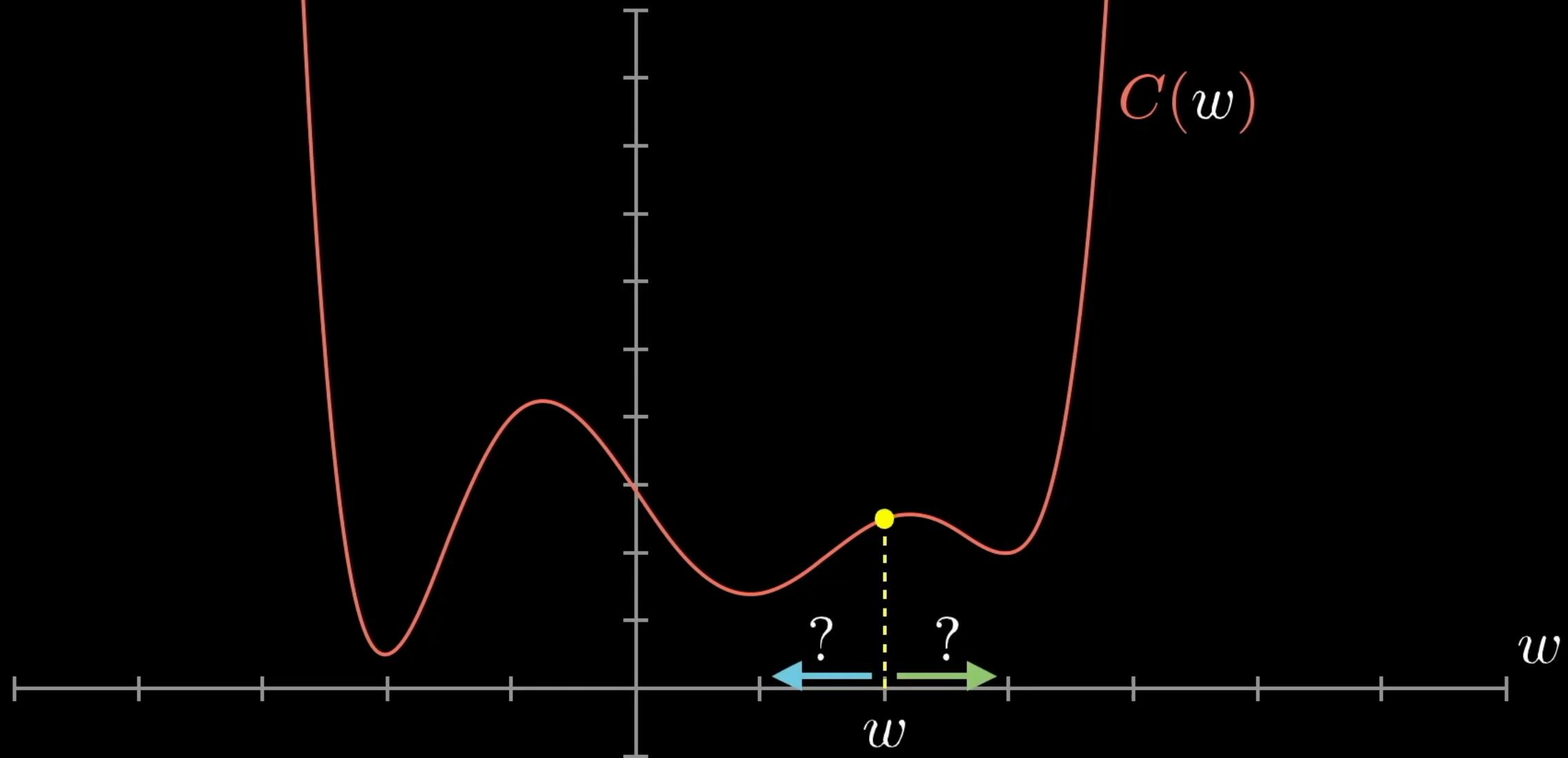
α : learning rate

t : training step

$\nabla f(x)$: gradient

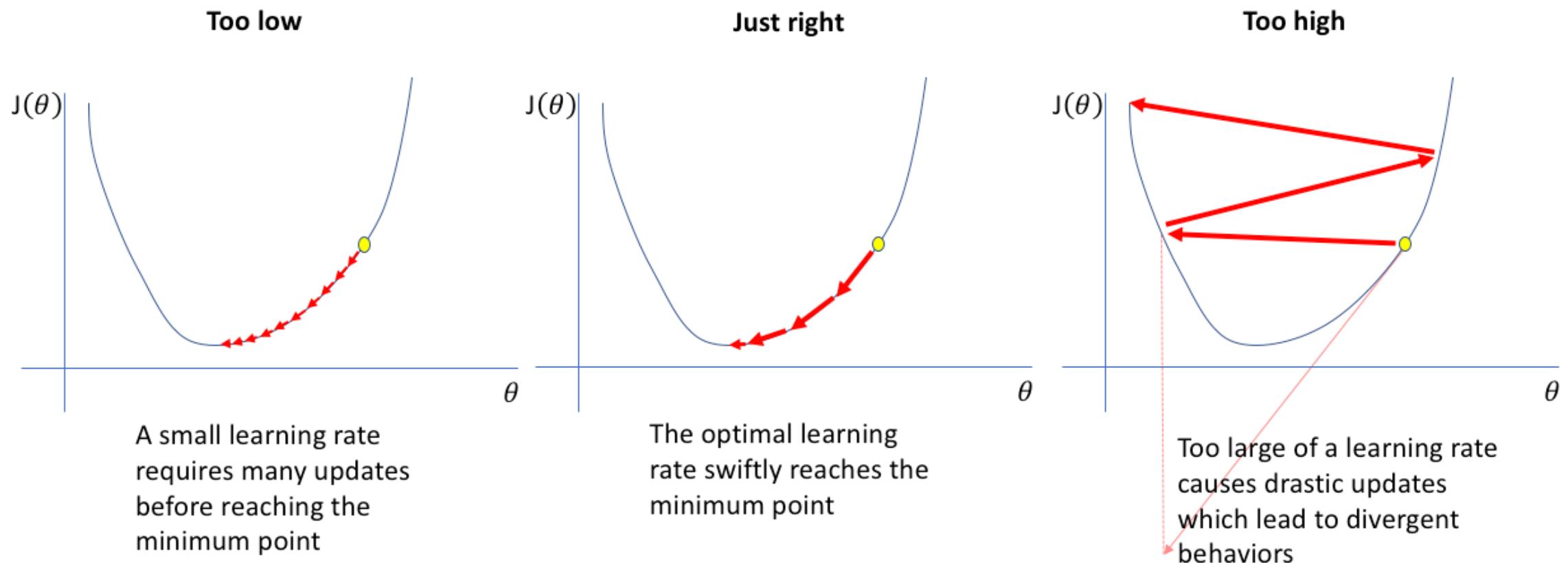
Why do we need gradient descent? Why not just taking the derivative of $J(\theta)$ and setting it to zero to find the minimum (like what we did for linear regression)?

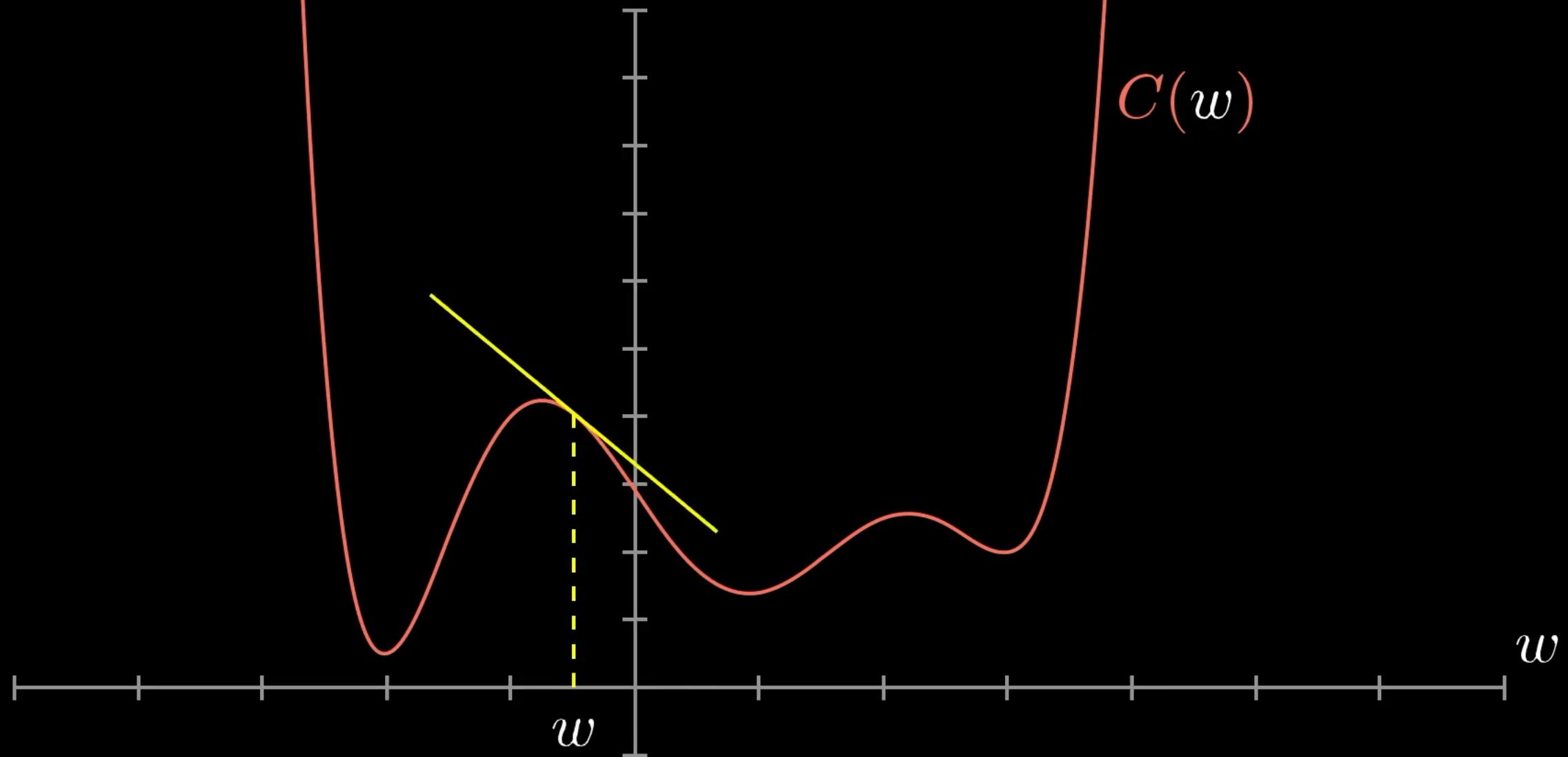
$$\min_{\theta} J(\theta) = \min_{\theta} \sum_{i=1}^n L(y^{(i)}, \hat{y}^{(i)})$$



We can reach a local minimum of the loss function by following the gradient (i.e., slope).

We need to adjust the learning rate strategically. A **large learning rate** could lead to **divergent behavior** in the model. For example, the training loss could wiggle.

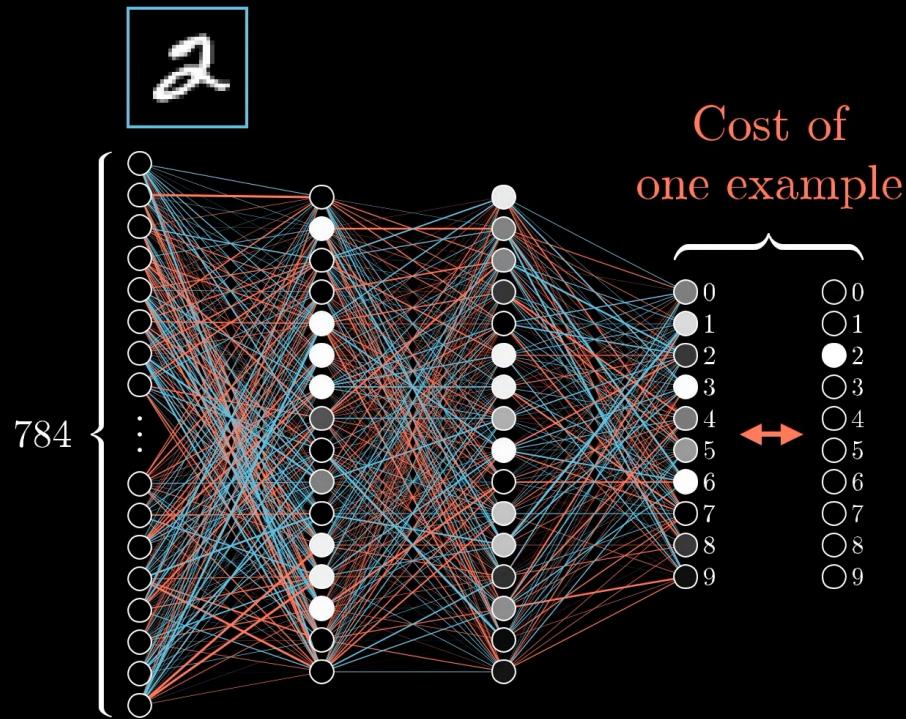




Take smaller steps (i.e., smaller learning rate) when the slope is getting small.

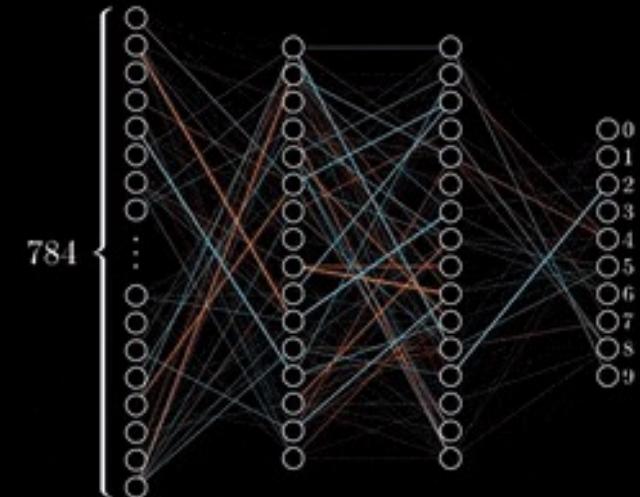
How do we adjust the weight after computing the loss for each iteration? To update the weights in previous layers, we need to use the **backpropagation** algorithm.

Compute the loss (error) for each iteration



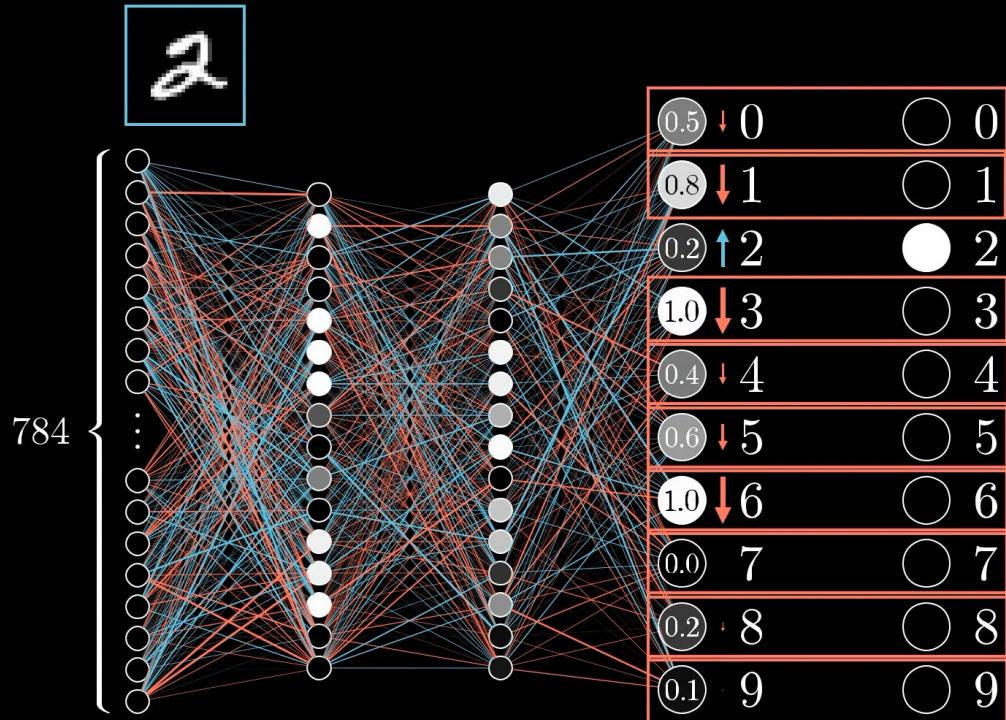
Backpropagate the errors

Training in progress...

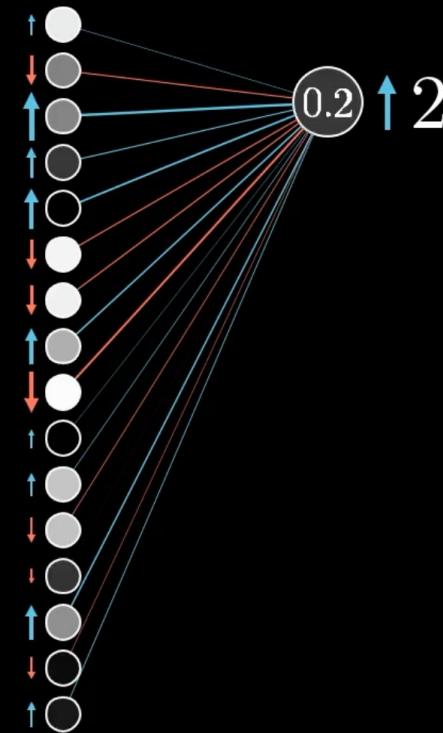


Intuitively speaking, after comparing the prediction and the ground truth, we want to **increase the weight** for the neuron we care about the most and decrease the others.

Compare the prediction and ground truth

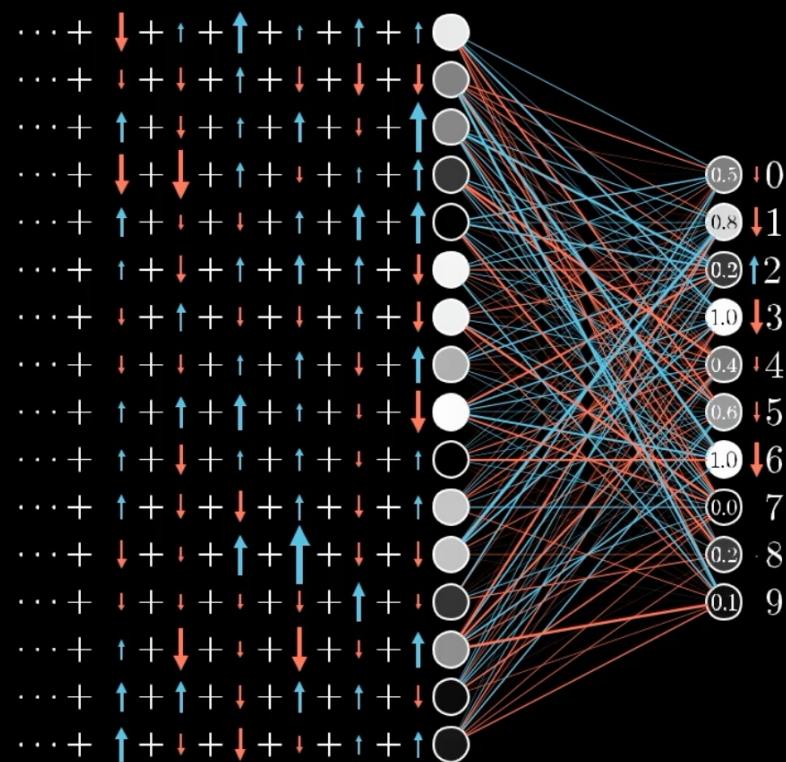


Update the weight for a neuron

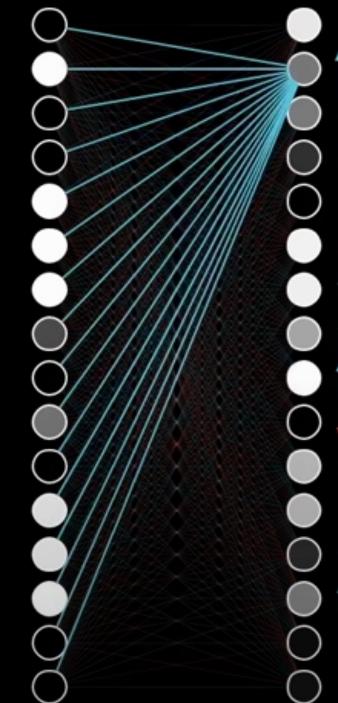


We apply the same idea to **iteratively update all the weights** for all the neurons in every previous layer, starting from the last layer, and **backpropagate the errors** back.

Update the weights for the current layer



Update the weights for the previous layer

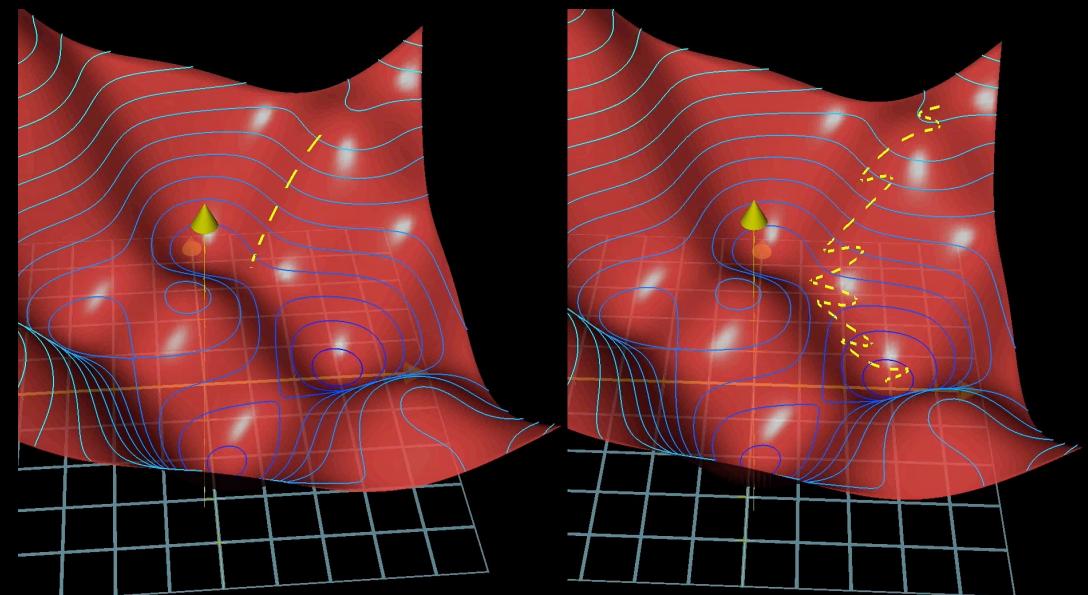


In practice, we use **mini-batches** (instead of all data) when running gradient descent to increase the speed (and save computer memory) when updating neuron weights.

Divide data randomly into mini-batches



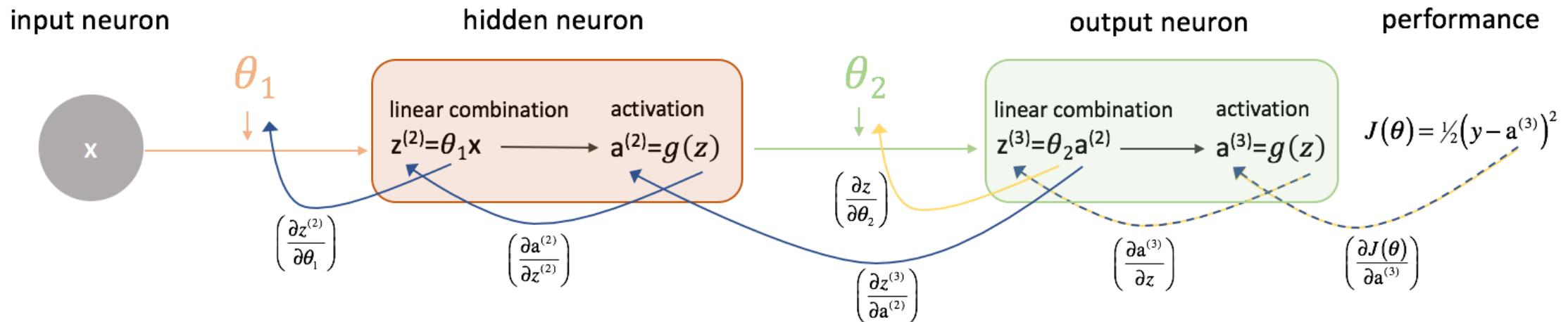
Use mini-batches to approximate the original gradient



The backpropagation algorithm applies the **chain rule** in calculus to compute gradient.

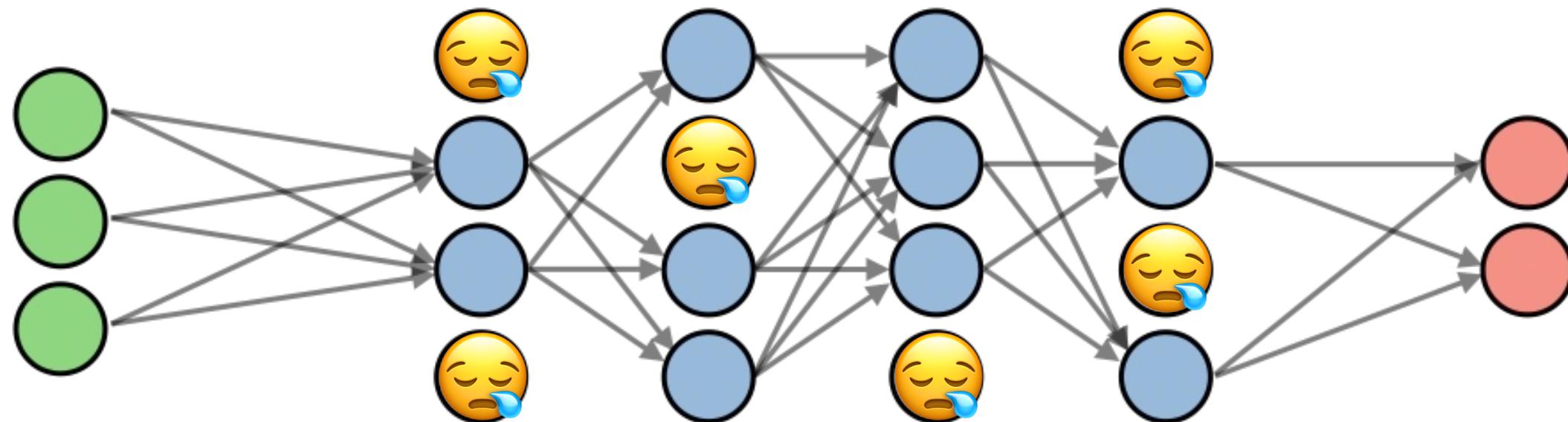
$$\frac{\partial g(f(\theta))}{\partial \theta} = \frac{\partial g}{\partial f} \cdot \frac{\partial f}{\partial \theta}$$

$$\nabla J(\theta_1) = \frac{\partial J(\theta)}{\partial \theta_1} = \frac{\partial J(\theta)}{\partial a^{(3)}} \cdot \frac{\partial a^{(3)}}{\partial z^{(3)}} \cdot \frac{\partial z^{(3)}}{\partial a^{(2)}} \cdot \frac{\partial a^{(2)}}{\partial z^{(2)}} \cdot \frac{\partial z^{(2)}}{\partial \theta_1}$$

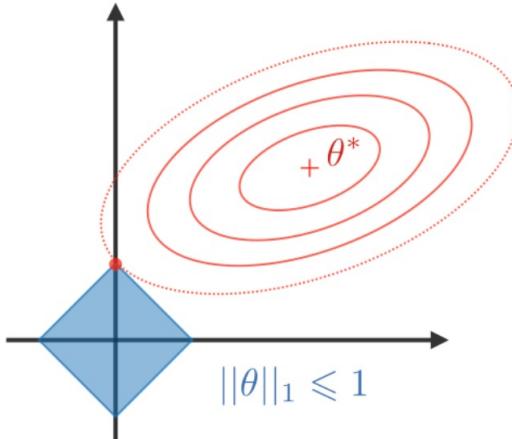
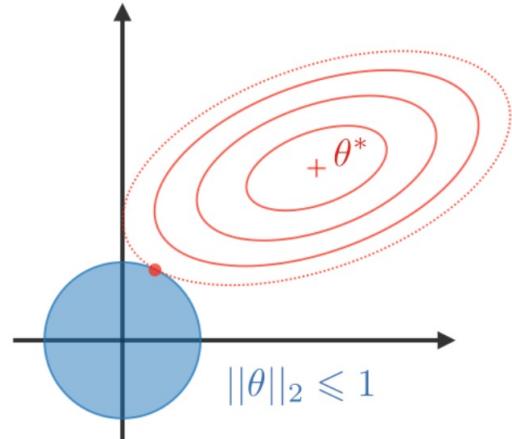
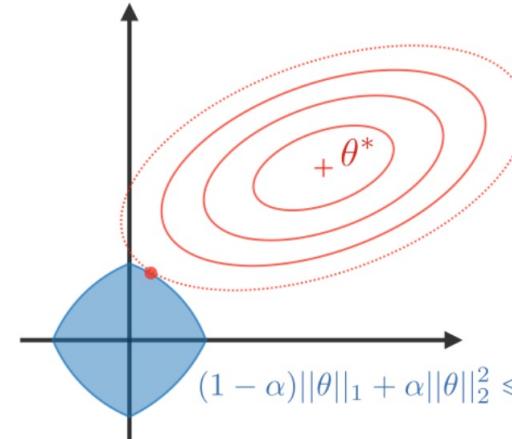


Deep neural nets can overfit easily due to a huge number of parameters. How can we deal with overfitting?

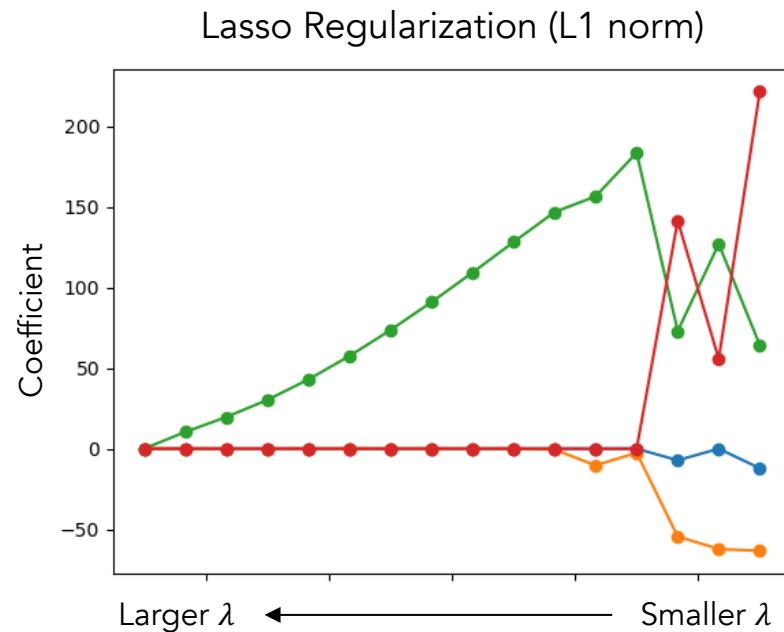
We can combat overfitting by randomly **dropping out neurons** with a pre-defined probability (i.e., the dropout technique), which forces the model to avoid paying too much attention to a particular set of features.



We can also combat overfitting by using the **regularization** technique (also setting its strength factor λ), which regulates model weights to ensure that they are not too large.

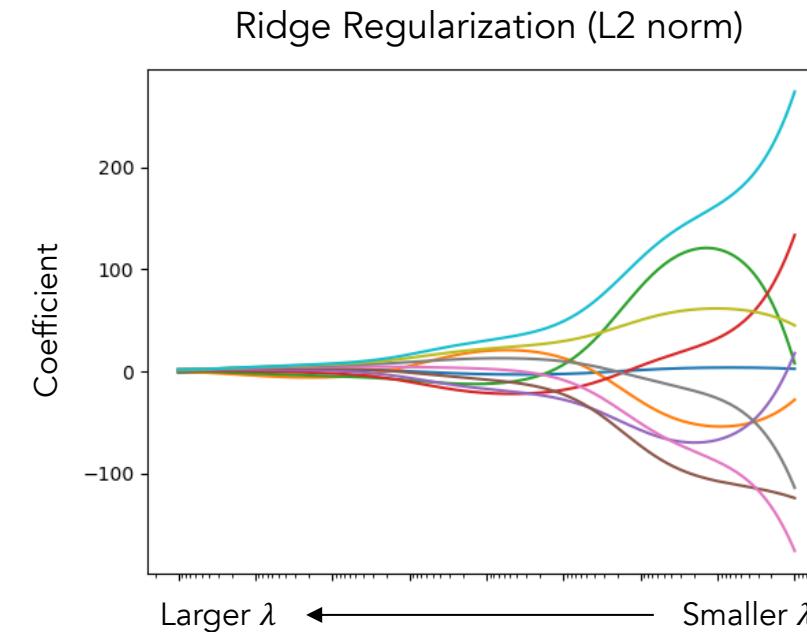
LASSO	Ridge	Elastic Net
<ul style="list-style-type: none"> • Shrinks coefficients to 0 • Good for variable selection  $ \theta _1 \leq 1$	<p>Makes coefficients smaller</p>  $ \theta _2 \leq 1$	<p>Tradeoff between variable selection and small coefficients</p>  $(1 - \alpha) \theta _1 + \alpha \theta _2 \leq 1$
$\dots + \lambda \theta _1$ $\lambda \in \mathbb{R}$	$\dots + \lambda \theta _2^2$ $\lambda \in \mathbb{R}$	$\dots + \lambda \left[(1 - \alpha) \theta _1 + \alpha \theta _2^2 \right]$ $\lambda \in \mathbb{R}, \alpha \in [0, 1]$

Increasing regularization strength λ shrinks the coefficients. Lasso specifically shrinks the coefficients to zero (which can be used for feature selection), while Ridge only shrink the coefficients to very small numbers that are close to zero (but not exactly zero).



$$\min_{\theta} J(\theta) + \lambda \|\theta\|_1$$

$$\|\theta\|_1 = |\theta_1| + |\theta_2| + \dots + |\theta_p|$$



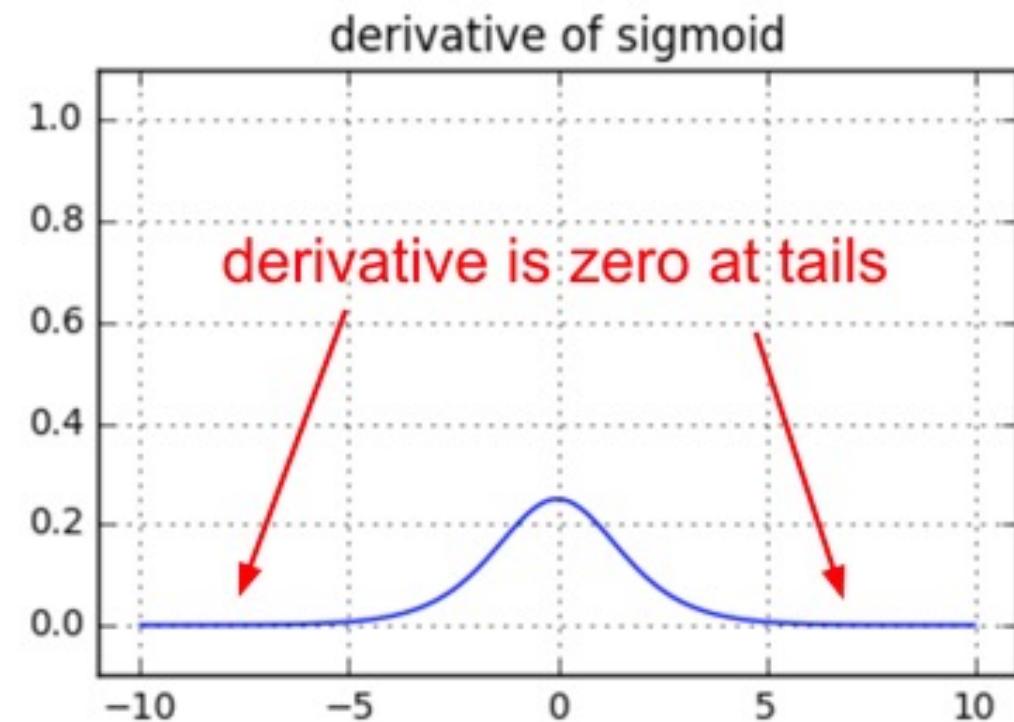
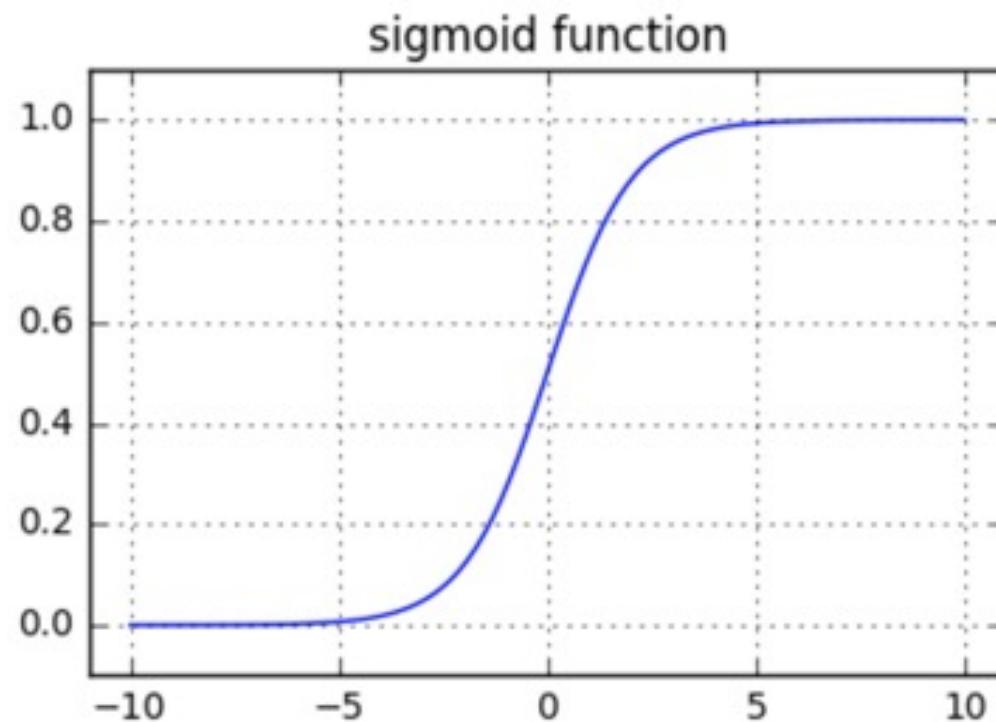
$$\min_{\theta} J(\theta) + \lambda \|\theta\|_2$$

$$\|\theta\|_2 = \sqrt{(\theta_1)^2 + (\theta_2)^2 + \dots + (\theta_p)^2}$$

We can also **augment our data on the fly** to combat overfitting (i.e., the so-called data augmentation technique), such as randomly rotating, cropping, and changing colors.

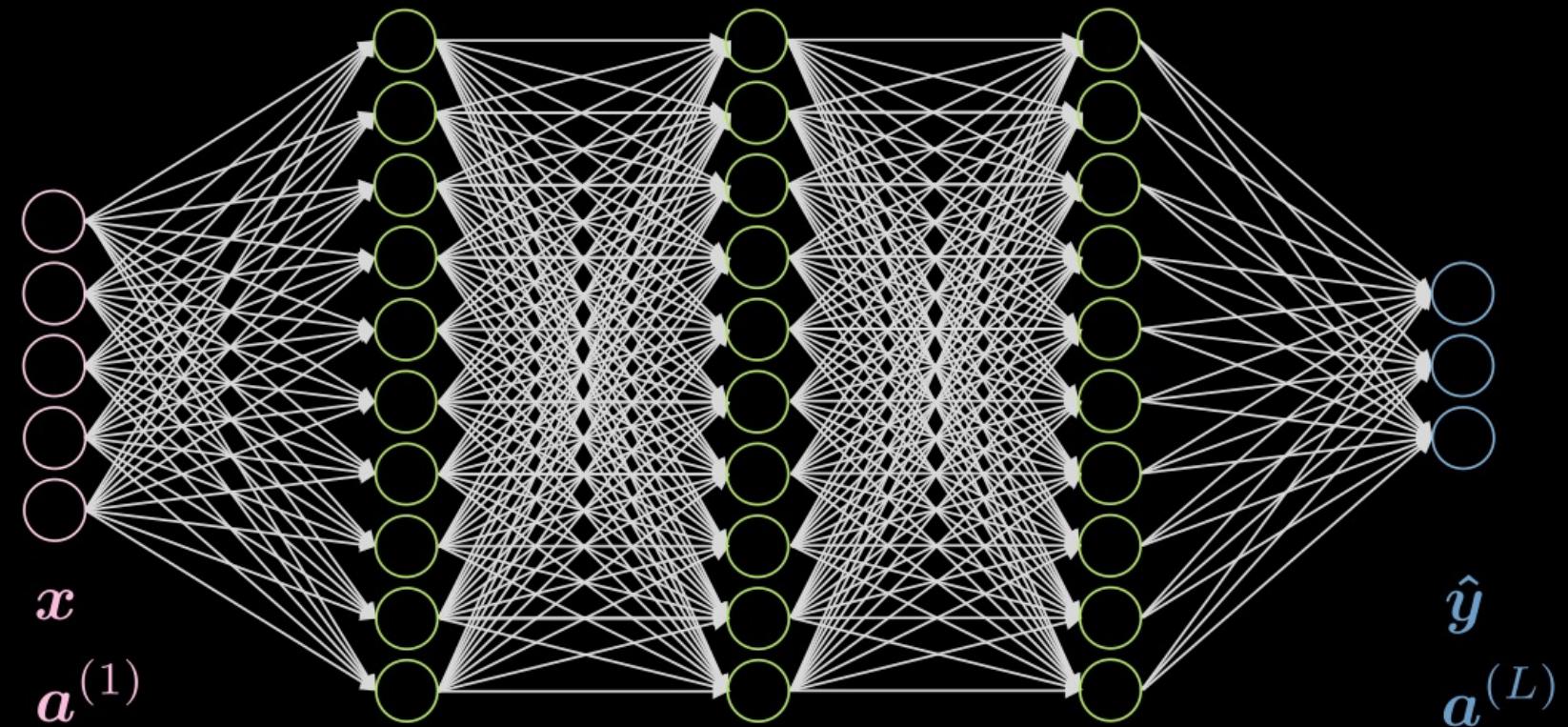
	Original	Sub-policy 1	Sub-policy 2	Sub-policy 3	Sub-policy 4	Sub-policy 5
Batch 1						
Batch 2						
Batch 3						
	ShearX, 0.9, 7 Invert, 0.2, 3	ShearY, 0.7, 6 Solarize, 0.4, 8	ShearX, 0.9, 4 AutoContrast, 0.8, 3	Invert, 0.9, 3 Equalize, 0.6, 3	ShearY, 0.8, 5 AutoContrast, 0.7, 3	

Deep learning models can suffer from **vanishing gradient**, where the gradient becomes too small during backpropagation, and thus the model weights are hard to update. The example below shows the problem when using the sigmoid activation function.



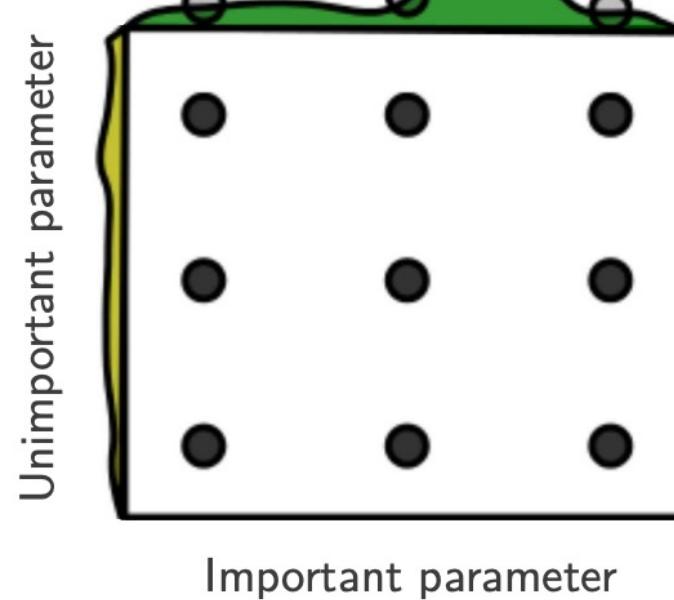
During model training, we need to tune **hyperparameters**, which means the parameters that are set prior to the learning process (unlike model parameters, i.e., model weights).

- Learning rate?
- Batch size?
- Layer size?
- Number of layers?
- Dropout rate?
- Strength of regularization?
- Activation function?
- Number of epoch?
- ...

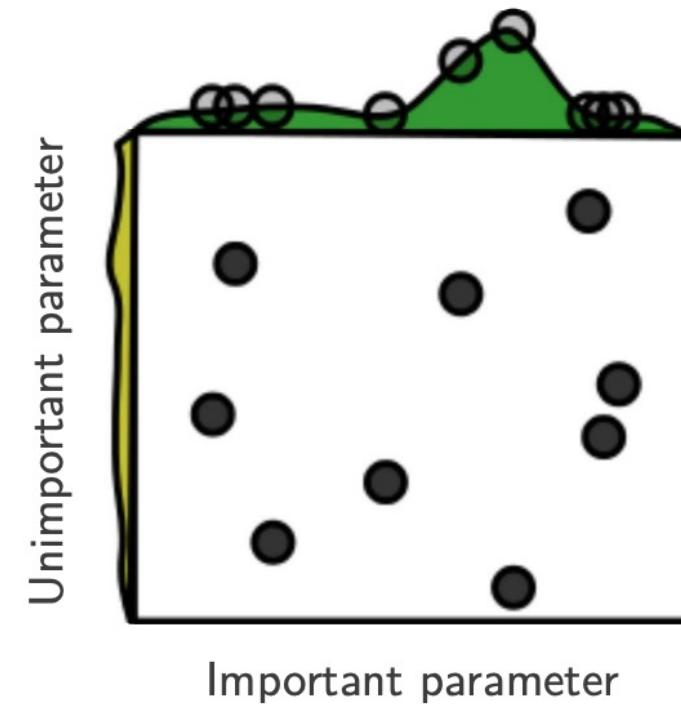


Common hyperparameter tuning strategies involve **grid or random search**. Random search is typically considered more effective when there are a lot of hyperparameters.

Grid Layout



Random Layout



Take-Away Messages

- Deep learning is the idea of stacking many layers of artificial neurons.
- You can change the activation function and loss function to perform different tasks.
- The activation function can transform the weighted sum of the input non-linearly.
- The loss function measures the distance between the ground truth and the prediction.
- Gradient descent is used to help us find a local minimum on the error (cost) function.
- When performing gradient descent, we need to set a learning rate to determine the pace.
- We need to use the backpropagation algorithm to iteratively update the weights in previous layers.
- Using dropout, regularization, or data augmentation can help us combat overfitting.



Questions?