Artificial Intelligence for Medicine II

Spring 2025

Lecture 112: NNs and Deep Learning

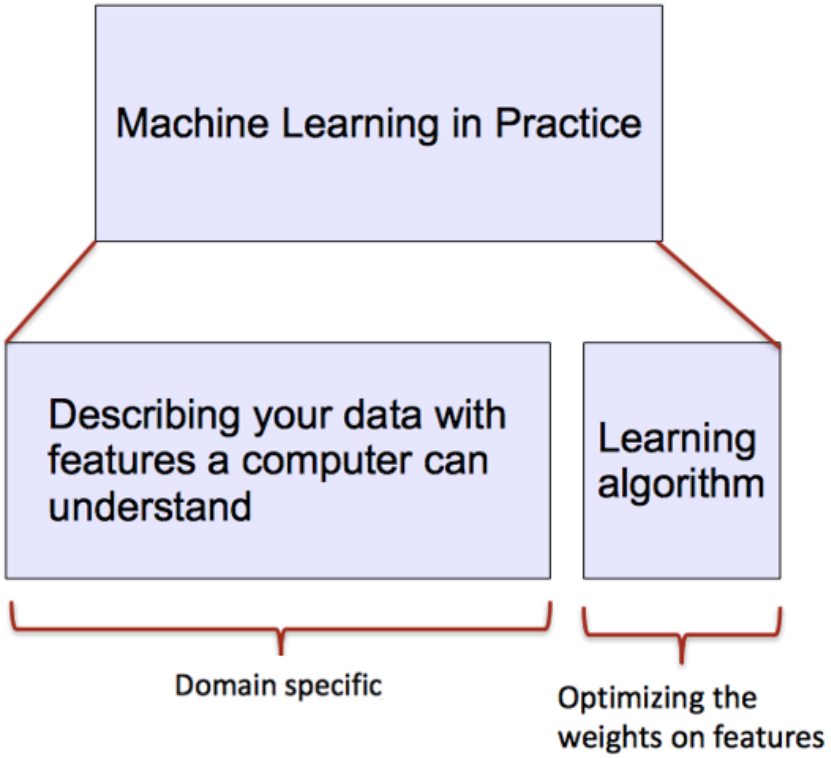
(Many slides adapted from mostly Alex Vakanski, D. Jurafsky, Bing Liu, Han, Kamber & Pei; Tan, Steinbach, Kumar and the web)

Lecture Outline

- ML vs. Deep Learning
- Introduction to NNs (Neural Networks)
- NN architectures
  - Convolutional NNs
  - Recurrent NNs
  - Encoder-Decoder NNs
  - Transformers

ML vs. Deep Learning

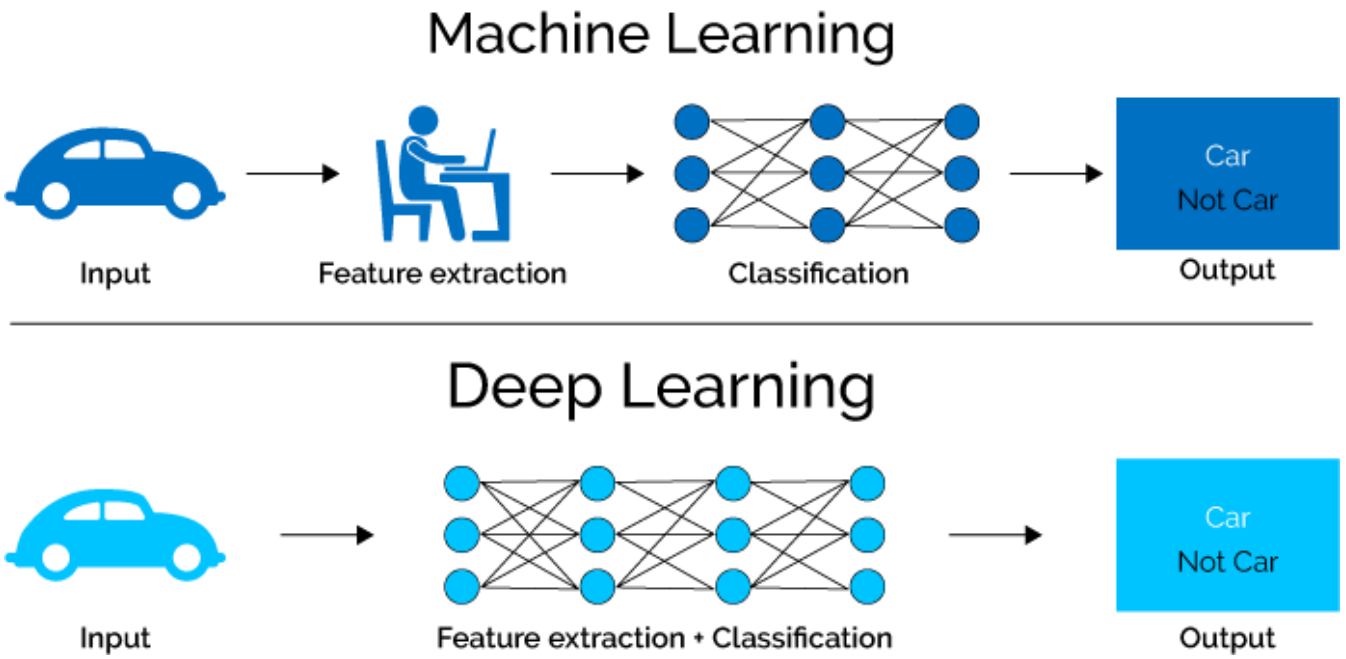
- Conventional machine learning methods rely on human-designed feature representations
  - ML becomes just optimizing weights to best make a final prediction



Picture from: Ismini Lourentzou – Introduction to Deep Learning

ML vs. Deep Learning

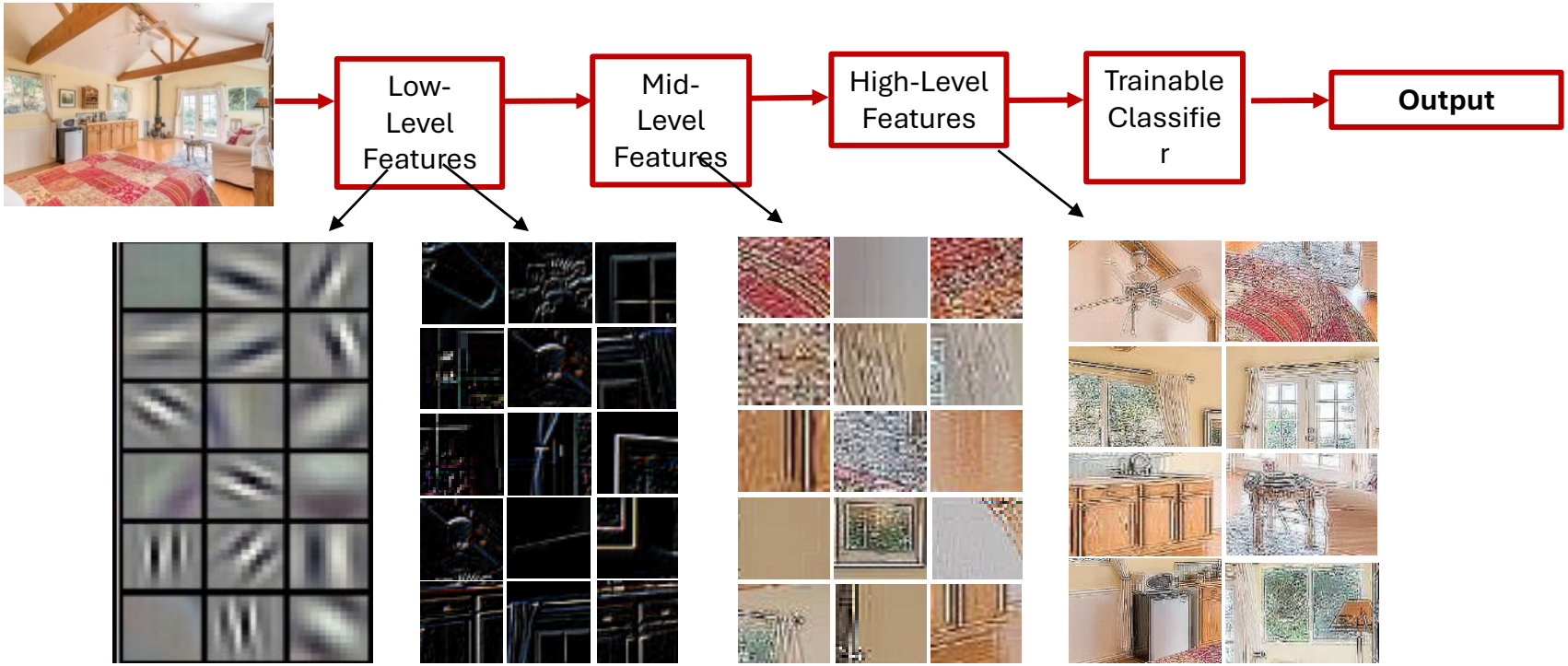
- ***Deep learning*** (DL) is a machine learning subfield that uses multiple layers for learning data representations
  - DL is exceptionally effective at learning patterns



Picture from: https://www.xenonstack.com/blog/static/public/uploads/media/machine-learning-vs-deep-learning.png

ML vs. Deep Learning

- DL applies a multi-layer process for learning rich hierarchical features (i.e., data representations)
  - Input image pixels → Edges → Textures → Parts → Objects



Slide credit: Param Vir Singh – Deep Learning

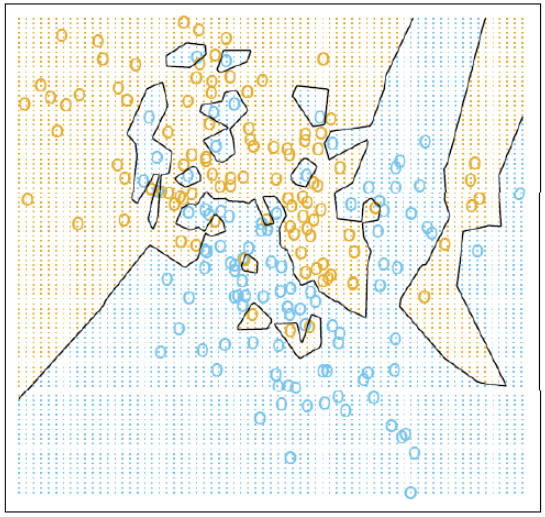
Why is DL Useful?

- DL provides a flexible, learnable framework for representing visual, text, linguistic information
  - Can learn in supervised and unsupervised manner
- DL represents an effective end-to-end learning system
- Requires large amounts of training data
- Since about 2010, DL has outperformed other ML techniques
  - First in vision and speech, then NLP, and other applications

Representational Power

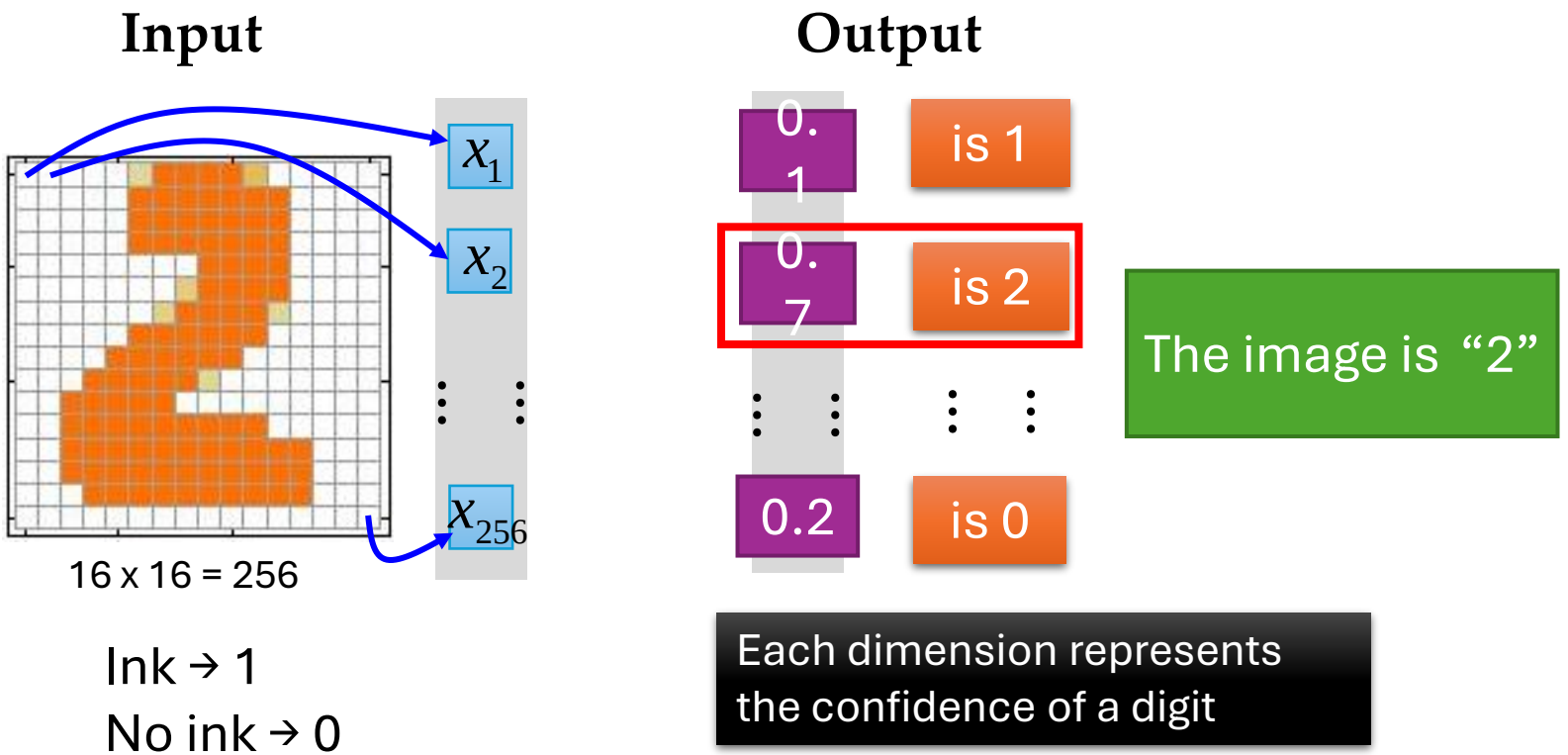
- NNs with at least one hidden layer are universal approximators
  - Given any continuous function $h(x)$ and some $\epsilon > 0$, there exists a NN with one hidden layer (and with a reasonable choice of non-linearity) described with the function $f(x)$, such that $\forall x, |h(x) - f(x)| < \epsilon$
  - I.e., NN can approximate any arbitrary complex continuous function

- NNs use nonlinear mapping of the inputs x to the outputs $f(x)$ to compute complex decision boundaries
- But then, why use deeper NNs?
  - The fact that deep NNs work better is an empirical observation
  - Mathematically, deep NNs have the same representational power as a one-layer NN



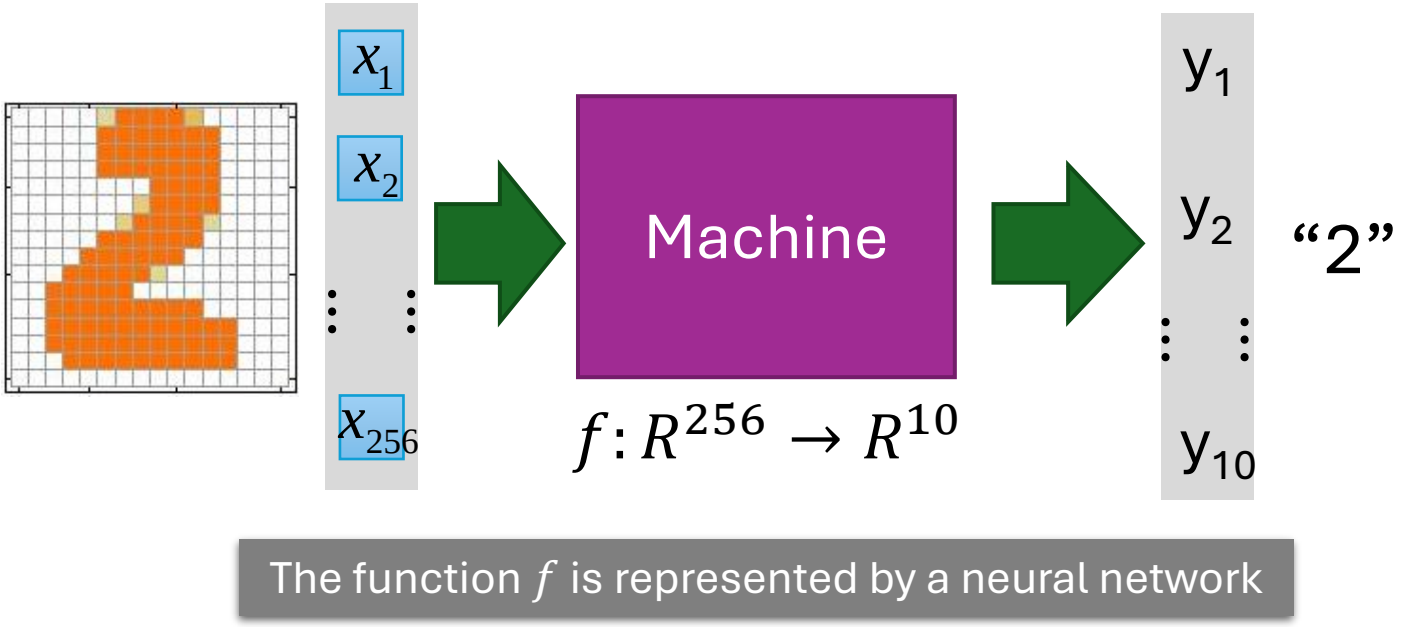
Introduction to Neural Networks

- Handwritten digit recognition (MNIST dataset)
  - The intensity of each pixel is considered an input element
  - Output is the class of the digit



Introduction to Neural Networks

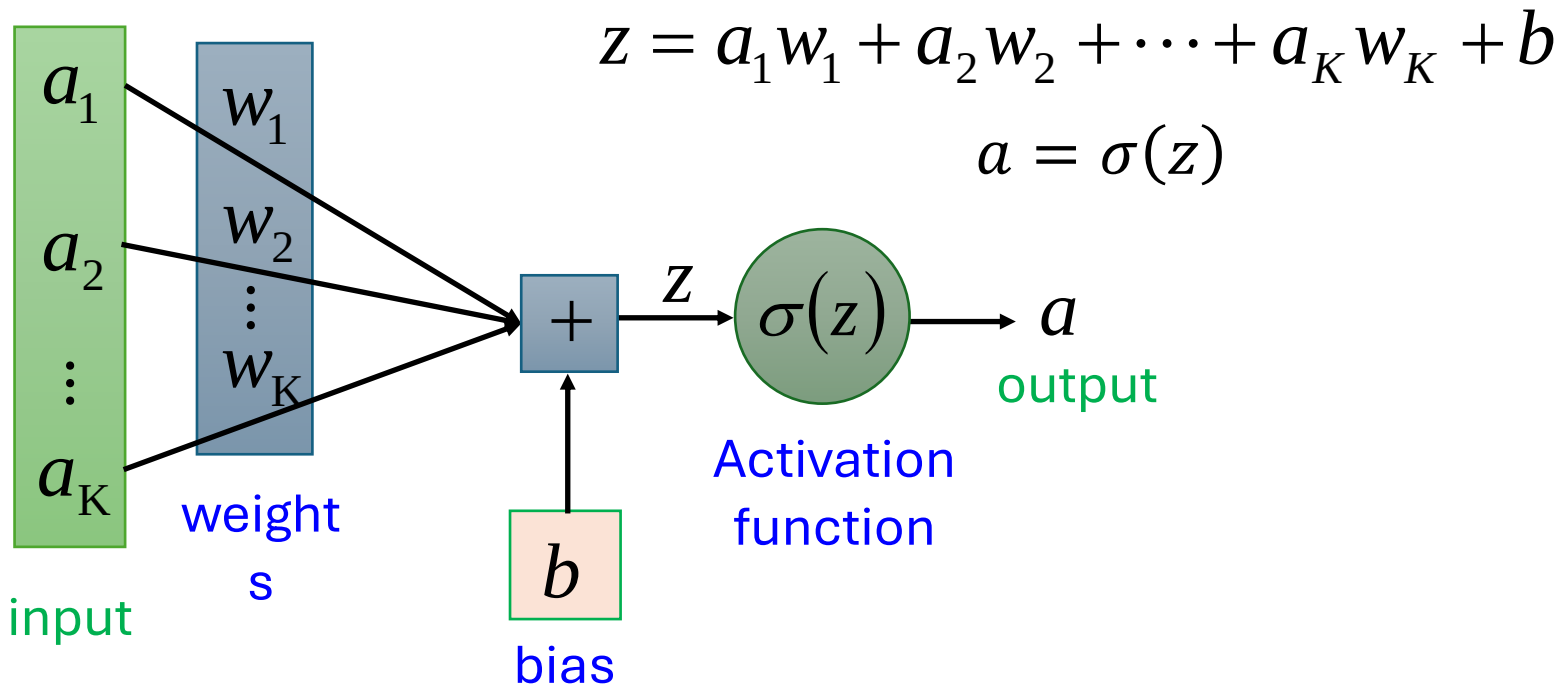
- Handwritten digit recognition



The function f is represented by a neural network

Elements of Neural Networks

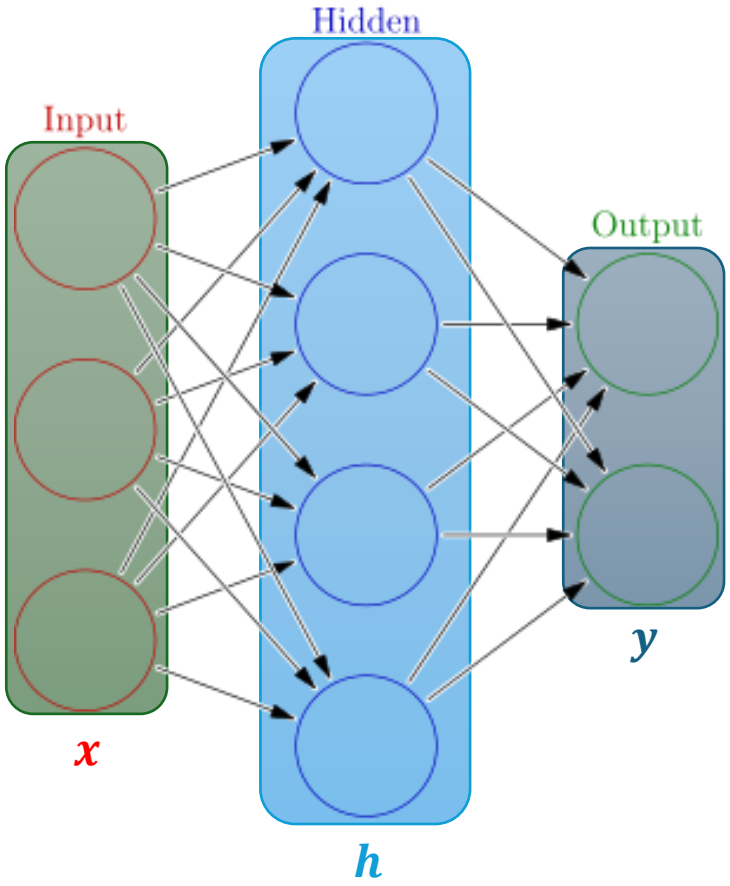
- NNs consist of hidden layers with neurons (i.e., computational units)
- A single neuron maps a set of inputs into an output number, or $f: R^K \rightarrow R$



Slide credit: Hung-yi Lee – Deep Learning Tutorial

Elements of Neural Networks

- A NN with one hidden layer and one output layer



Weights Biases

$$hidden\ layer\ h = \sigma(W_1 x + b_1)$$

$$output\ layer\ y = \sigma(W_2 h + b_2)$$

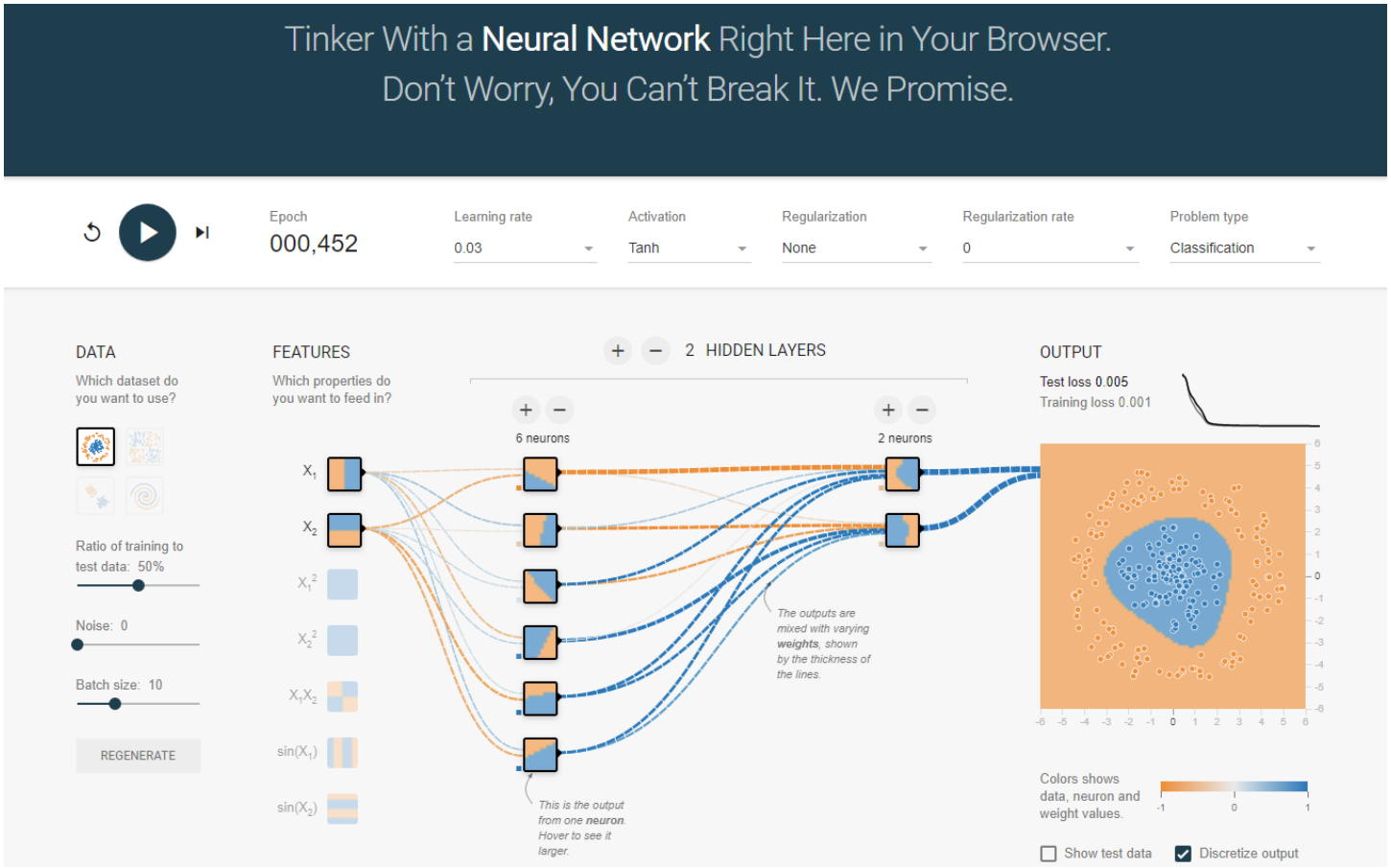
Activation functions

4 + 2 = 6 neurons (not counting inputs)
[3 × 4] + [4 × 2] = 20 weights
4 + 2 = 6 biases
26 learnable parameters

Slide credit: Ismini Lourentzou – Introduction to Deep Learning

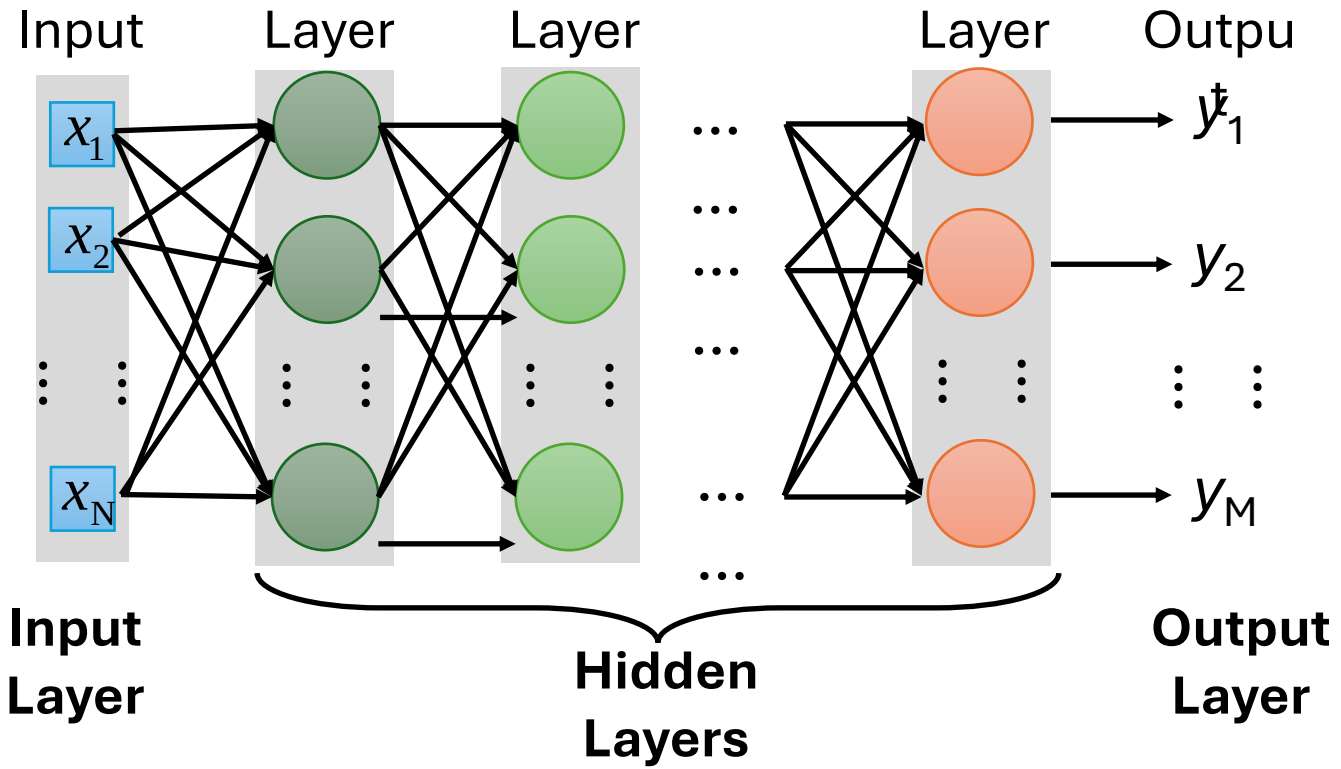
Elements of Neural Networks

- A neural network playground [link]



Elements of Neural Networks

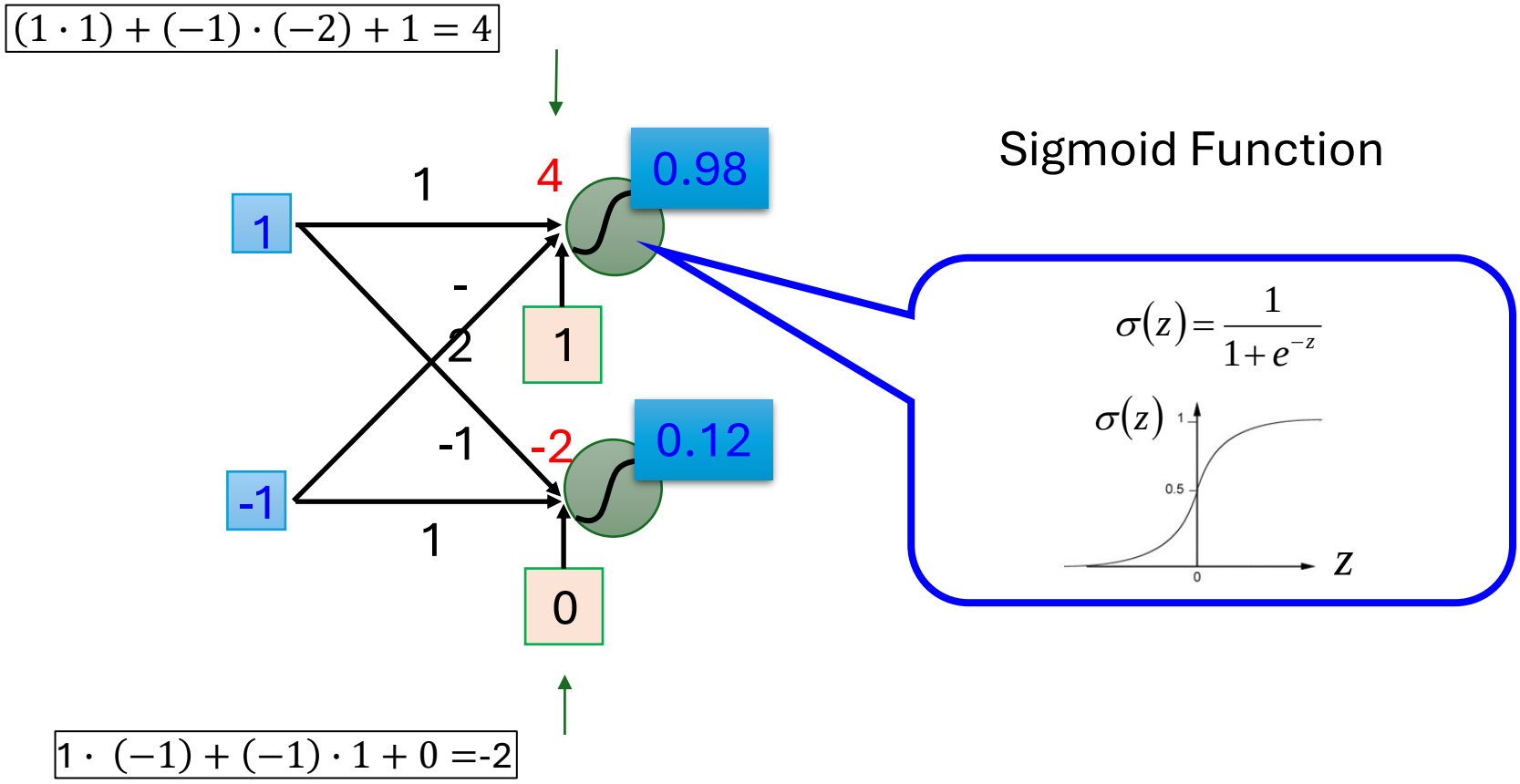
- Deep NNs have many hidden layers
  - Fully-connected (dense) layers (a.k.a. Multi-Layer Perceptron or MLP)
  - Each neuron is connected to all neurons in the succeeding layer



Slide credit: Hung-yi Lee – Deep Learning Tutorial

Elements of Neural Networks

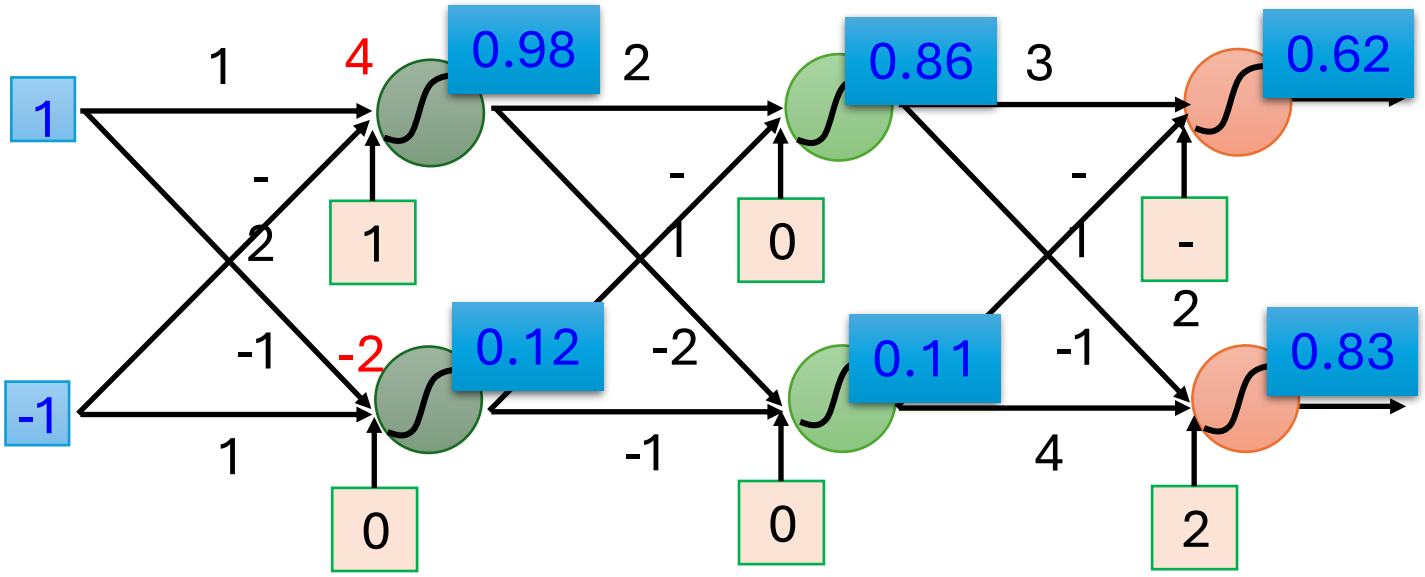
- A simple network, toy example



Slide credit: Hung-yi Lee – Deep Learning Tutorial

Elements of Neural Networks

- A simple network, toy example (cont'd)
  - For an input vector $[1 \quad -1]^T$, the output is $[0.62 \quad 0.83]^T$

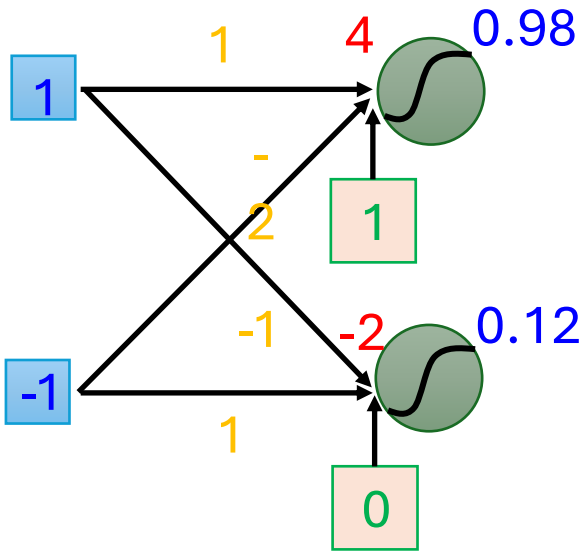


$$f: R^2 \rightarrow R^2 \qquad f\left(\begin{bmatrix}1\\-1\end{bmatrix}\right) = \begin{bmatrix}0.62\\0.83\end{bmatrix}$$

Slide credit: Hung-yi Lee – Deep Learning Tutorial

Matrix Operation

- Matrix operations are helpful when working with multidimensional inputs and outputs



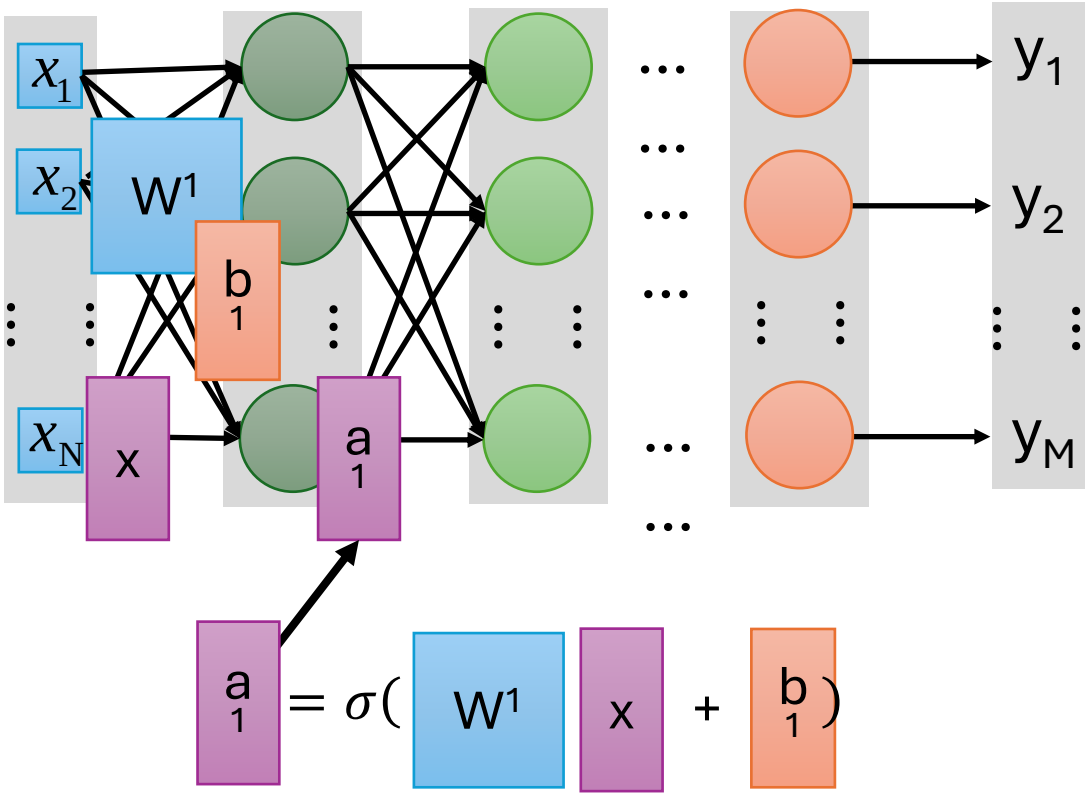
$$\sigma\left(\begin{bmatrix}1 & -2\\ -1 & 1\end{bmatrix}\begin{bmatrix}1\\ -1\end{bmatrix}+\begin{bmatrix}1\\ 0\end{bmatrix}\right)=\begin{bmatrix}0.98\\ 0.12\end{bmatrix}$$

$$\begin{bmatrix}4\\ -2\end{bmatrix}$$

$\sigma($ W x + b $) =$ a

Matrix Operation

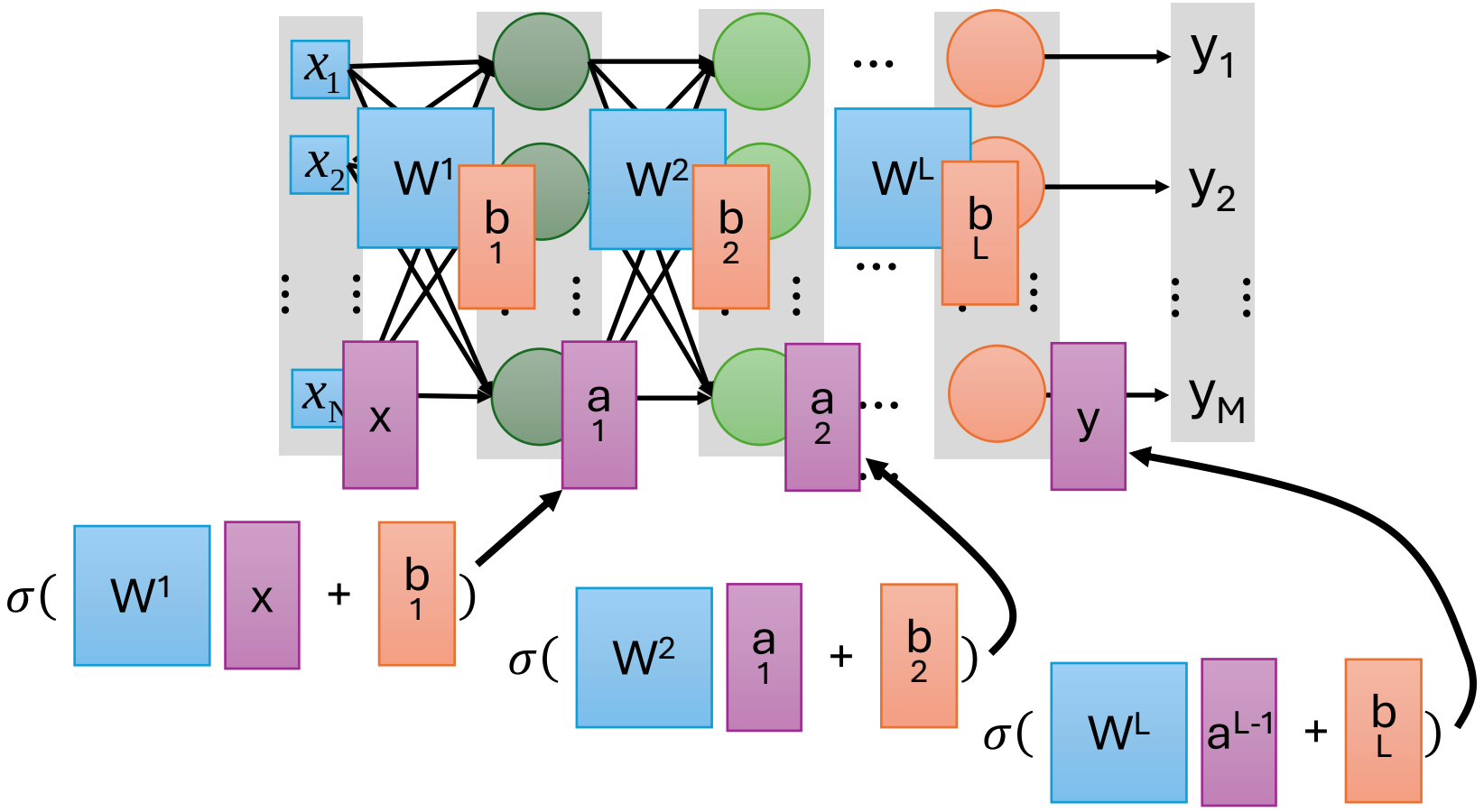
- Multilayer NN, matrix calculations for the first layer
  - Input vector x, weights matrix W^1, bias vector b^1, output vector a^1



Slide credit: Hung-yi Lee – Deep Learning Tutorial

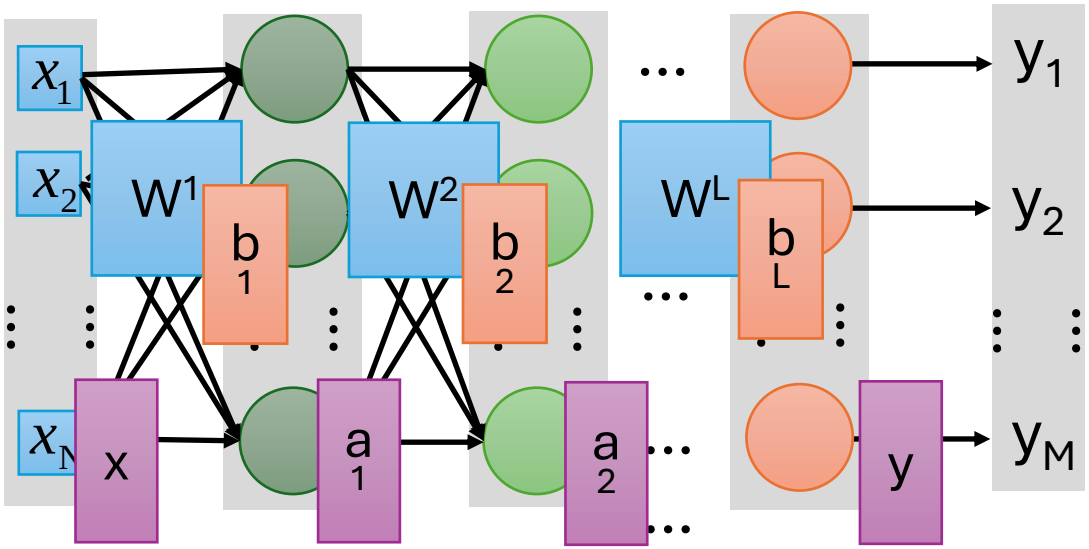
Matrix Operation

- Multilayer NN, matrix calculations for all layers



Matrix Operation

- Multilayer NN, function f maps inputs x to outputs y, i.e., $y = f(x)$

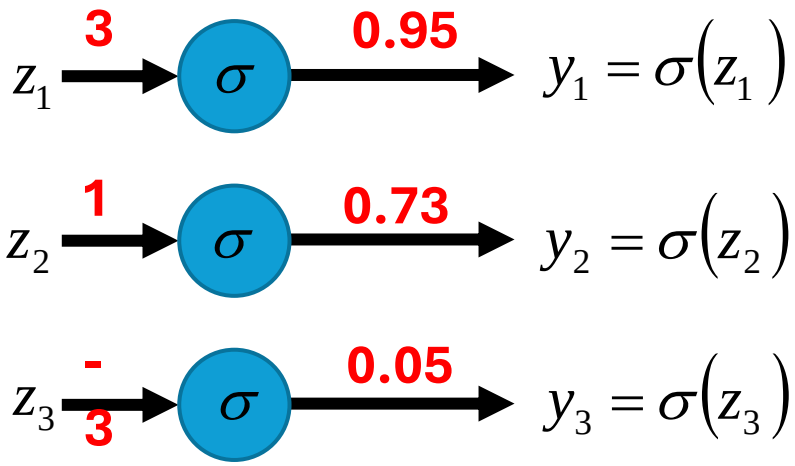


$$y = f(x) = \sigma(W^L \cdots \sigma(W^2 \sigma(W^1 x + b_1) + b_2) \cdots + b_L)$$

Softmax Layer

- In multi-class classification tasks, the output layer is typically a ***softmax layer***
  - I.e., it employs a ***softmax activation function***
  - If a layer with a sigmoid activation function is used as the output layer instead, the predictions by the NN may not be easy to interpret
    - Note that an output layer with sigmoid activations can still be used for binary classification

A Layer with Sigmoid Activations

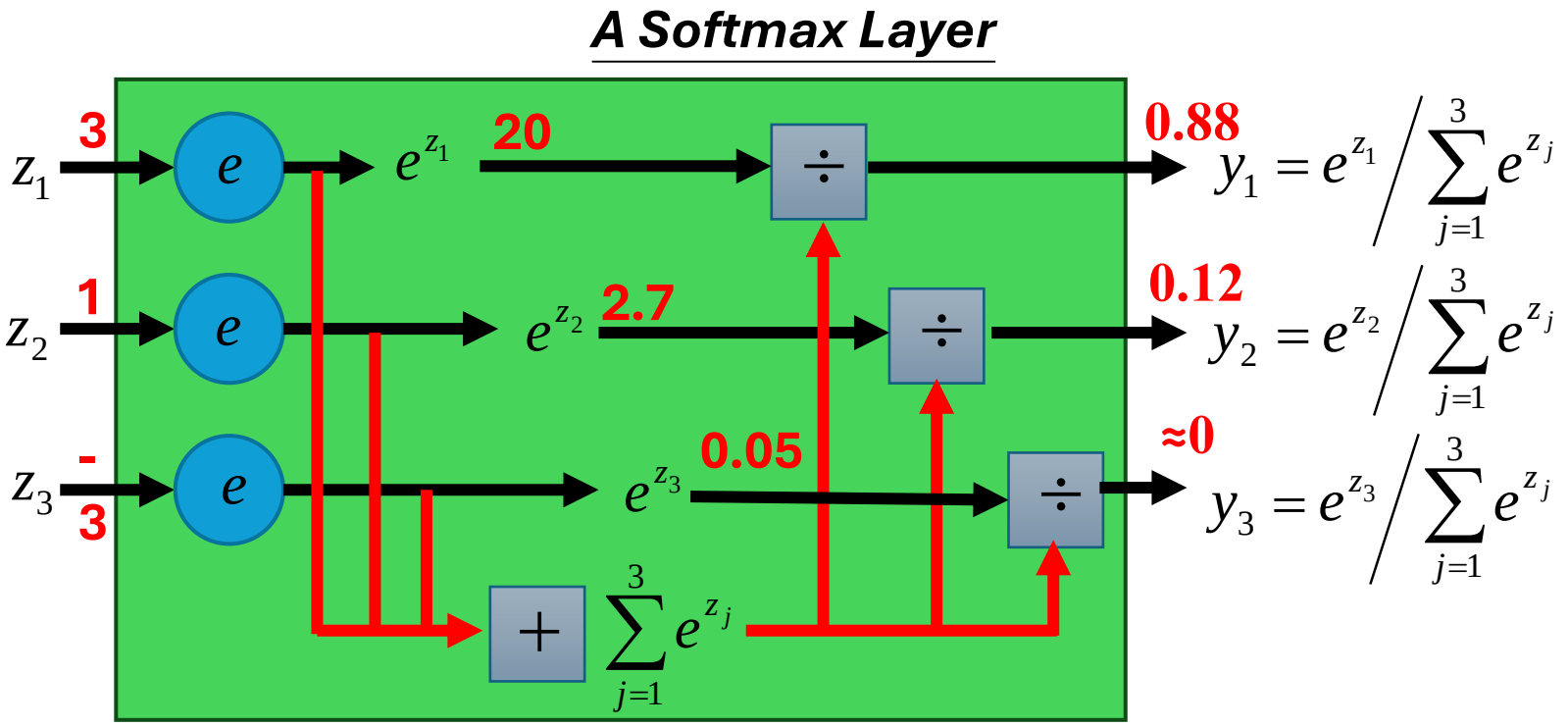


Slide credit: Hung-yi Lee – Deep Learning Tutorial

Softmax Layer

- The softmax layer applies softmax activations to output a probability value in the range [0, 1]
  - The values z inputted to the softmax layer are referred to as ***logits***

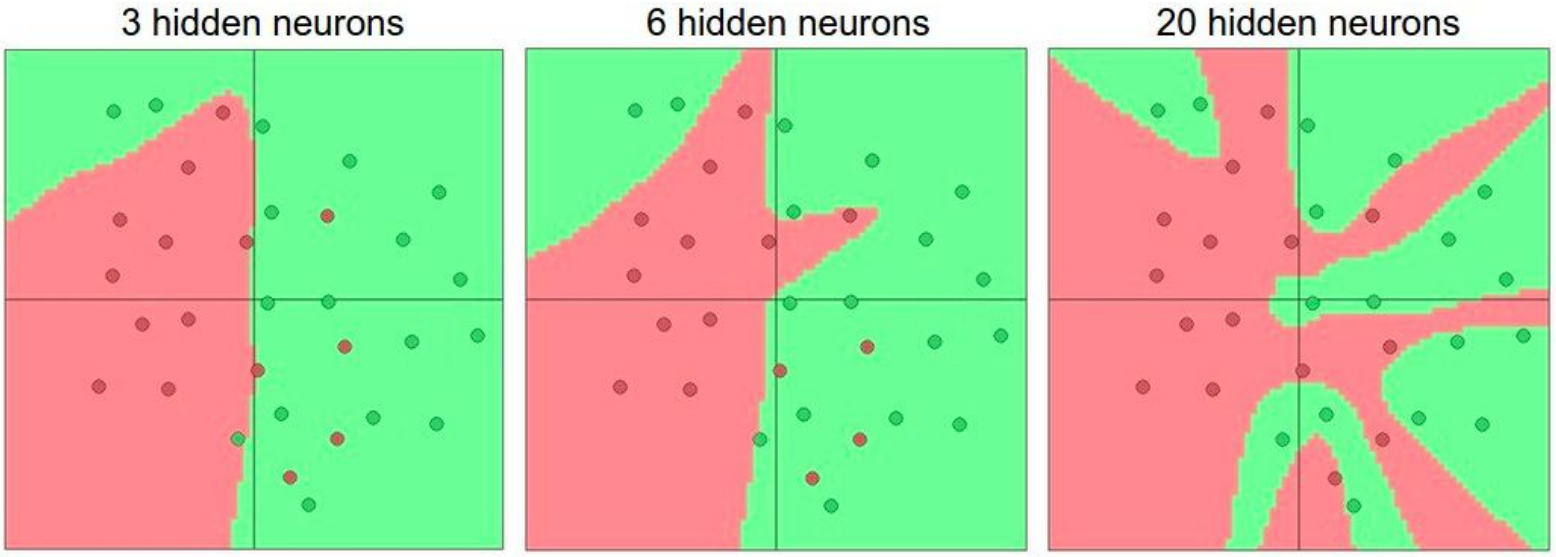
Probability:
- $0 < y_i < 1$
- $\sum_i y_i = 1$



Slide credit: Hung-yi Lee – Deep Learning Tutorial

Activation Functions

- Non-linear activations are needed to learn complex (non-linear) data representations
  - Otherwise, NNs would be just a linear function (such as $W_1 W_2 x = Wx$)
  - NNs with large number of layers (and neurons) can approximate more complex functions
    - Figure: more neurons improve representation (but, may overfit)

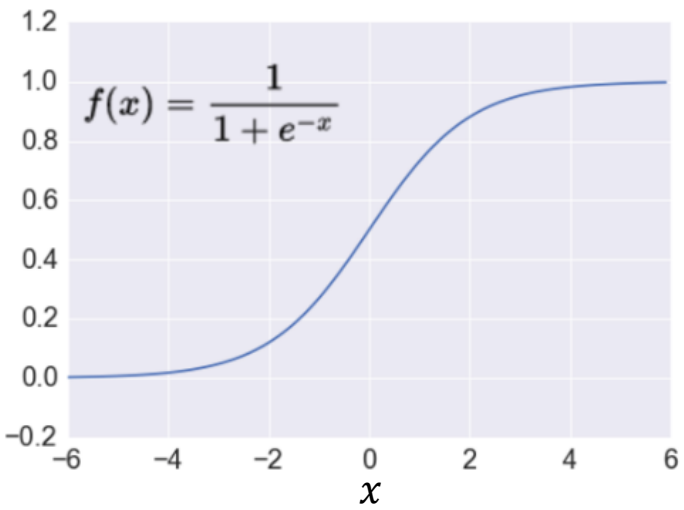


Picture from: http://cs231n.github.io/assets/nn1/layer_sizes.jpeg

Activation: Sigmoid

- ***Sigmoid function*** σ: takes a real-valued number and "squashes" it into the range between 0 and 1
  - The output can be interpreted as the firing rate of a biological neuron
    - Not firing = 0; Fully firing = 1
  - When the neuron's activation are 0 or 1, sigmoid neurons saturate
    - Gradients at these regions are almost zero (almost no signal will flow)
  - Sigmoid activations are less common in modern NNs

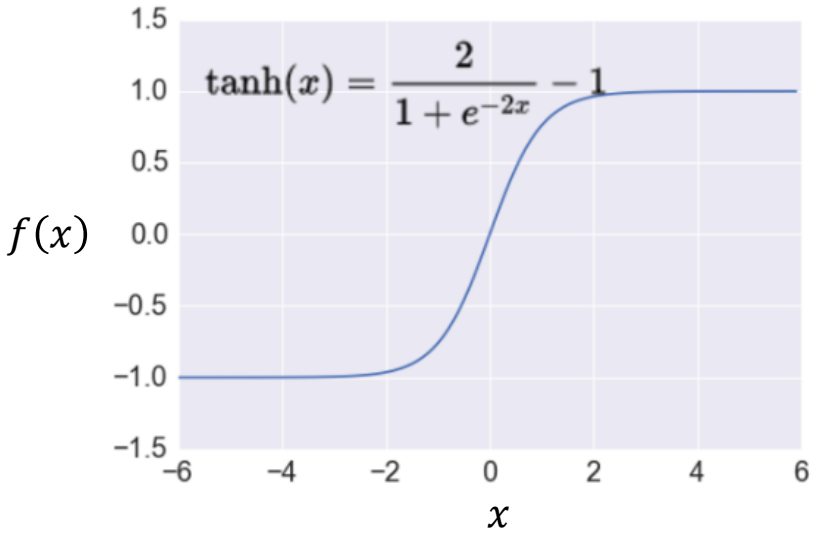
$f(x)$



$\mathbb{R}^n \rightarrow [0,1]$

Activation: Tanh

- ***Tanh function***: takes a real-valued number and "squashes" it into range between -1 and 1
  - Like sigmoid, tanh neurons saturate
  - Unlike sigmoid, the output is zero-centered
    - It is therefore preferred than sigmoid
  - Tanh is a scaled sigmoid: $\tanh(x) = 2 \cdot \sigma(2x) - 1$



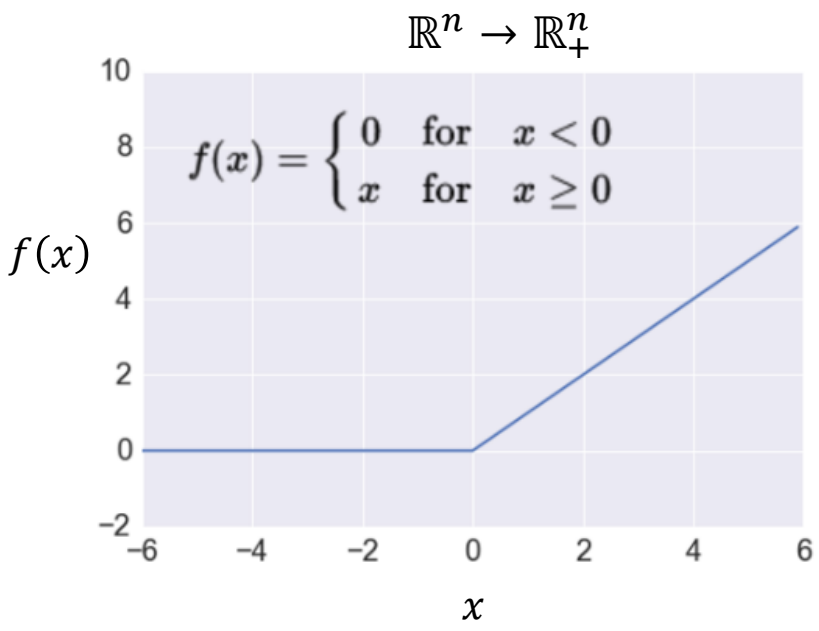
$\mathbb{R}^n \rightarrow [-1,1]$

Activation: ReLU

- ***ReLU*** (Rectified Linear Unit): takes a real-valued number and thresholds it at zero

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

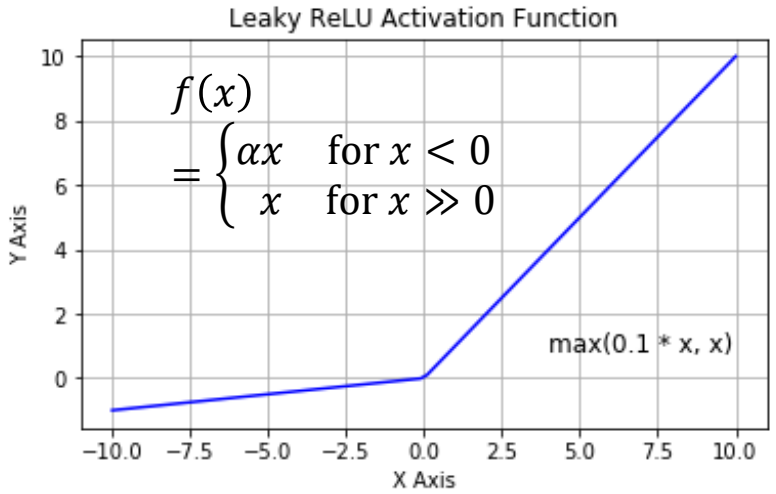
  - Most modern deep NNs use ReLU activations
  - ReLU is fast to compute
    - Compared to sigmoid, tanh
    - Simply threshold a matrix at zero
  - Accelerates the convergence of gradient descent
    - Due to linear, non-saturating form
  - Prevents the gradient vanishing problem



Activation: Leaky ReLU

- The problem of ReLU activations: they can "die"
  - ReLU could cause weights to update in a way that the gradients can become zero and the neuron will not activate again on any data
  - E.g., when a large learning rate is used

- ***Leaky ReLU*** activation function is a variant of ReLU
  - Instead of the function being 0 when $x < 0$, a leaky ReLU has a small negative slope (e.g., $\alpha = 0.01$, or similar)
    - This resolves the dying ReLU problem
    - Most current works still use ReLU
      - With a proper setting of the learning rate, the problem of dying ReLU can be avoided

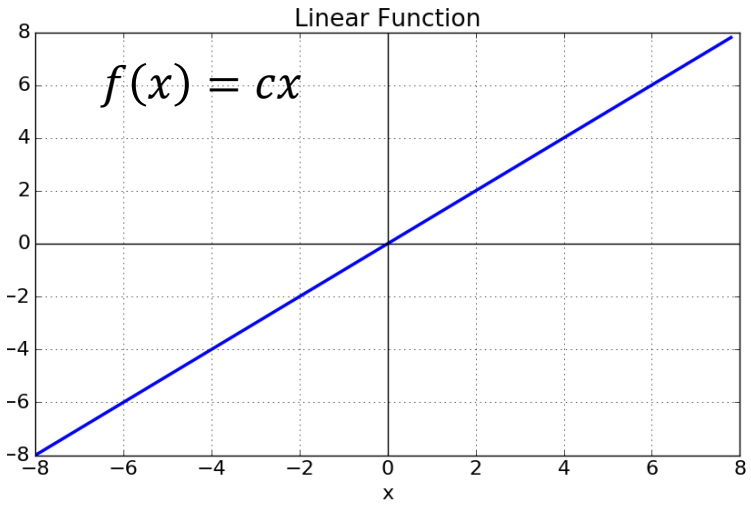


Activation: Linear Function

- ***Linear function*** means that the output signal is proportional to the input signal to the neuron

  - If the value of the constant c is 1, it is also called identity activation function
  - This activation type is used in regression problems
    - E.g., the last layer can have linear activation function, in order to output a real number (and not a class membership)

$$\mathbb{R}^n \rightarrow \mathbb{R}^n$$



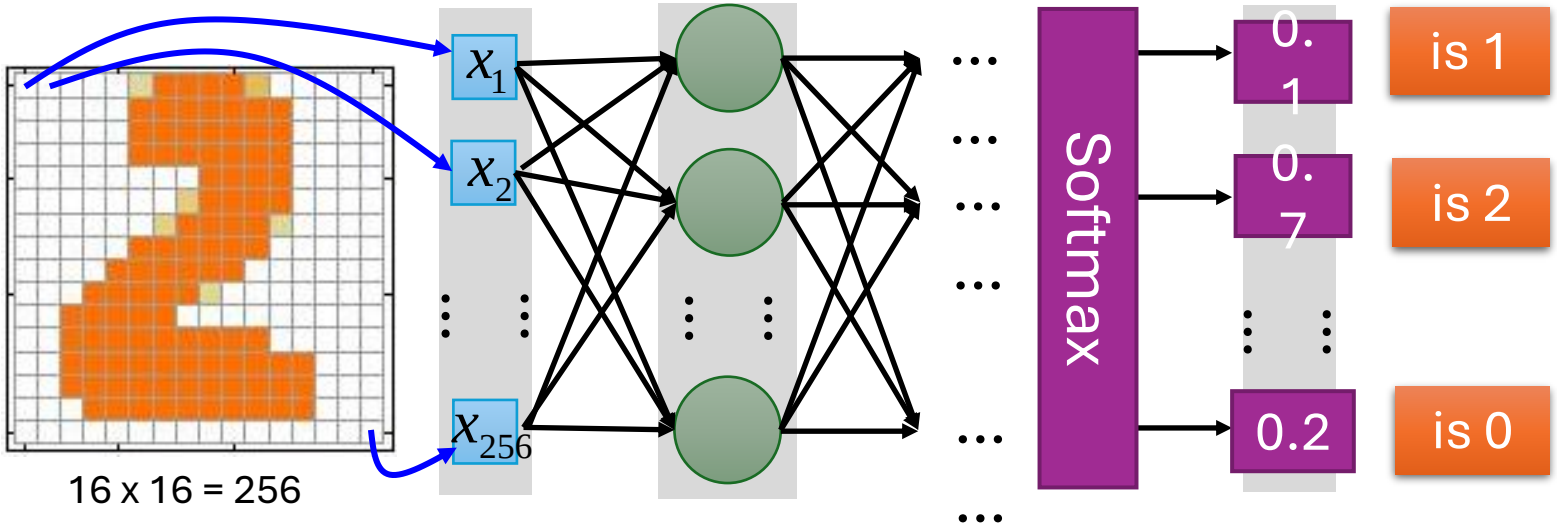
Training NNs

Training Neural Networks

- The network ***parameters*** θ include the weight matrices and bias vectors from all layers

$$\theta = \{W^1, b^1, W^2, b^2, \cdots W^L, b^L\}$$

  - Often, the model parameters θ are referred to as weights
- Training a model to learn a set of parameters θ that are optimal (according to a criterion) is one of the greatest challenges in ML

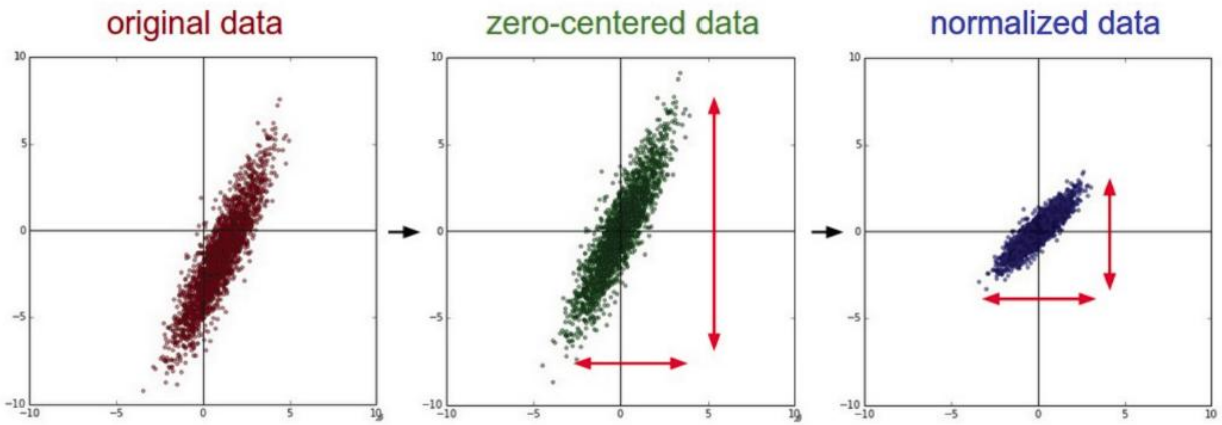


Slide credit: Hung-yi Lee – Deep Learning Tutorial

Training NNs

Training Neural Networks

- ***Data preprocessing*** - helps convergence during training
  - Mean subtraction, to obtain zero-centered data
    - Subtract the mean for each individual data dimension (feature)
  - Normalization
    - Divide each feature by its standard deviation
      - To obtain standard deviation of 1 for each data dimension (feature)
    - Or, scale the data within the range [0,1] or [-1, 1]
      - E.g., image pixel intensities are divided by 255 to be scaled in the [0,1] range

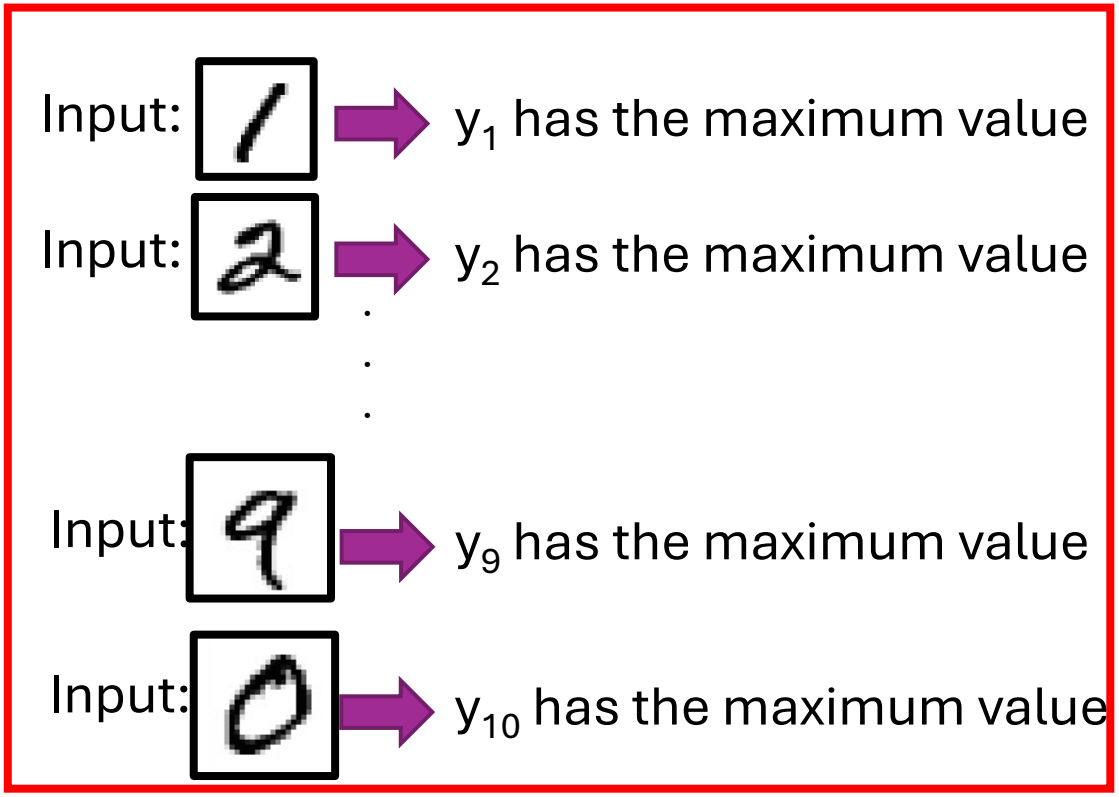


Picture from: https://cs231n.github.io/neural-networks-2/

Training NNs

Training Neural Networks

- To train a NN, set the parameters θ such that for a training subset of images, the corresponding elements in the predicted output have maximum values

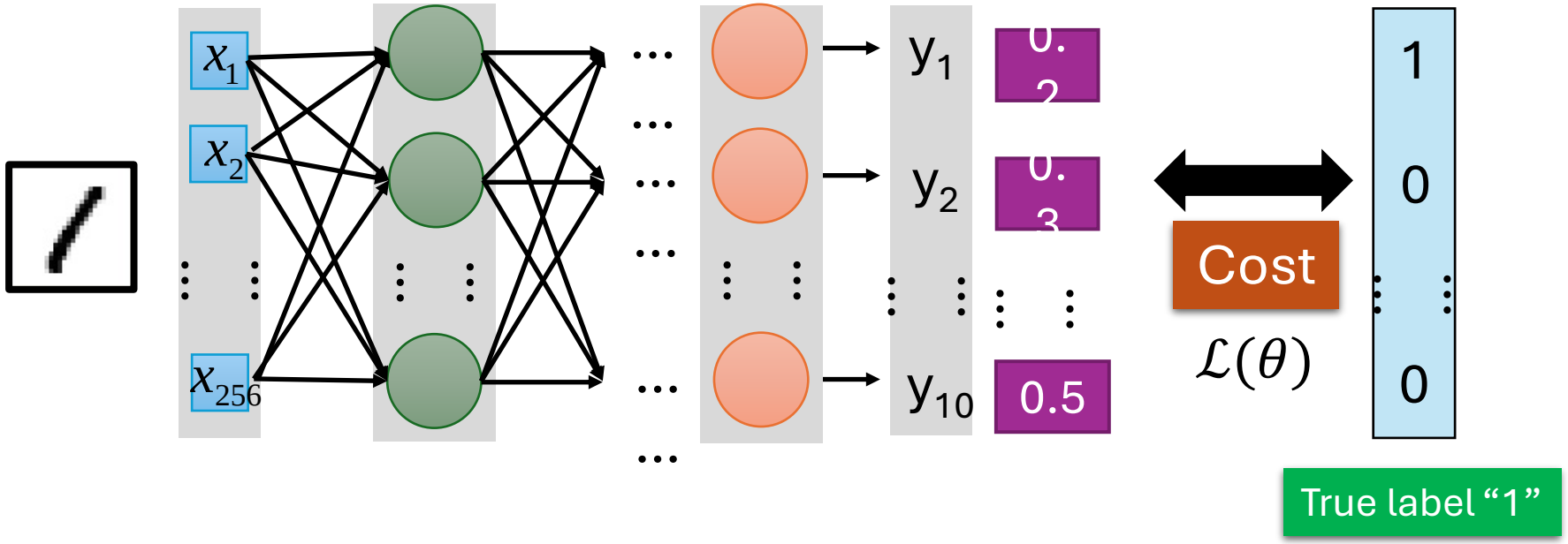


Slide credit: Hung-yi Lee – Deep Learning Tutorial

Training NNs

Training Neural Networks

- Define a ***loss function***/objective function/cost function $\mathcal{L}(\theta)$ that calculates the difference (error) between the model prediction and the true label
  - E.g., $\mathcal{L}(\theta)$ can be mean-squared error, cross-entropy, etc.

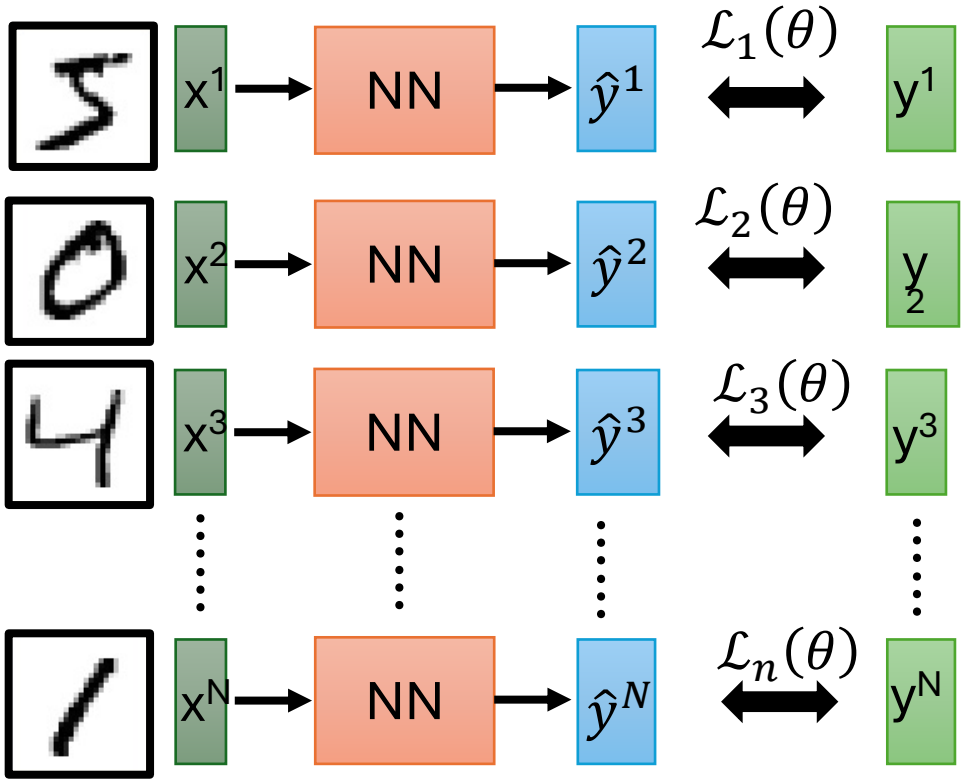


Slide credit: Hung-yi Lee – Deep Learning Tutorial

Training NNs

Training Neural Networks

- For a training set of N images, calculate the total loss overall all images: $\mathcal{L}(\theta) = \sum_{n=1}^{N} \mathcal{L}_n(\theta)$
- Find the optimal parameters θ^* that minimize the total loss $\mathcal{L}(\theta)$



Slide credit: Hung-yi Lee – Deep Learning Tutorial

Loss Functions

Training Neural Networks

- ***Classification tasks***

Training examples

Pairs of N inputs x_i and ground-truth class labels y_i

Output Layer

Softmax Activations
[maps to a probability distribution]

$$P(y = j \mid \mathbf{x}) = \frac{e^{\mathbf{x}^\top \mathbf{w}_j}}{\sum_{k=1}^{K} e^{\mathbf{x}^\top \mathbf{w}_k}}$$

Loss function

Cross-entropy

$$\mathcal{L}(\theta) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{K}\left[y_k^{(i)} \log \hat{y}_k^{(i)} + \left(1 - y_k^{(i)}\right) \log\left(1 - \hat{y}_k^{(i)}\right)\right]$$

Ground-truth class labels y_i and model predicted class labels $\hat{y}_i$

Slide credit: Ismini Lourentzou – Introduction to Deep Learning

Loss Functions

- ***Regression tasks***

Training examples

Pairs of N inputs x_i and ground-truth output values y_i

Output Layer

Linear (Identity) or Sigmoid Activation

Loss function

Mean Squared Error

$$\mathcal{L}(\theta) = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2$$

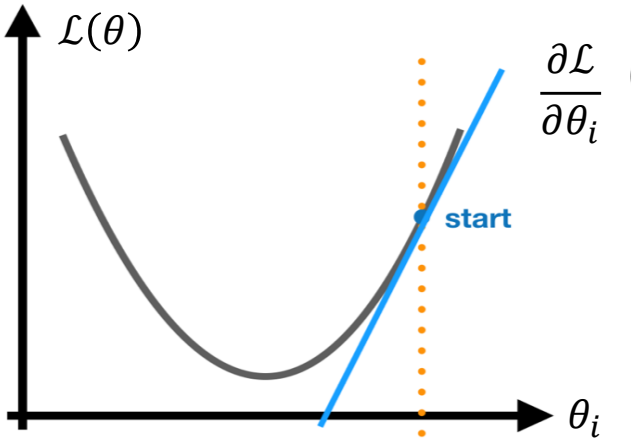
Mean Absolute Error

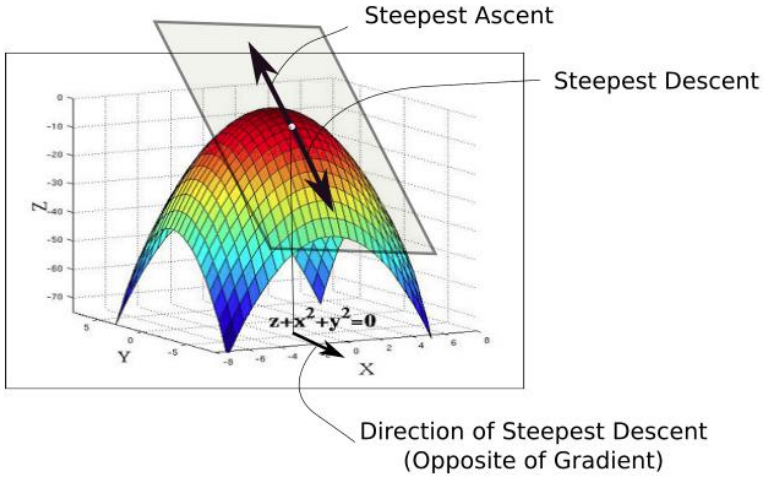
$$\mathcal{L}(\theta) = \frac{1}{n}\sum_{i=1}^{n}\left|y^{(i)} - \hat{y}^{(i)}\right|$$

Slide credit: Ismini Lourentzou – Introduction to Deep Learning

Training NNs

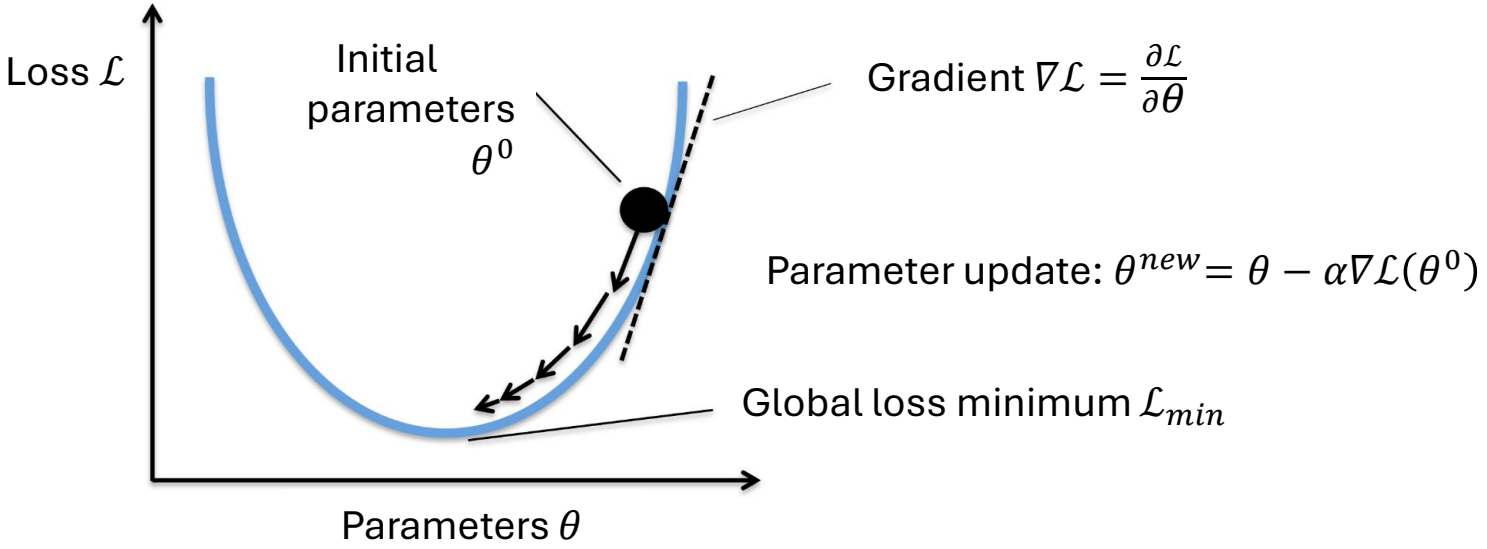
- Optimizing the loss function $\mathcal{L}(\theta)$
  - Almost all DL models these days are trained with a variant of the ***gradient descent*** (GD) algorithm
  - GD applies iterative refinement of the network parameters θ
  - GD uses the opposite direction of the gradient of the loss with respect to the NN parameters (i.e.,$\nabla\mathcal{L}(\theta) = [\partial\mathcal{L}/\partial\theta_i]$) for updating θ
    - The gradient of the loss function $\nabla\mathcal{L}(\theta)$ gives the direction of fastest increase of the loss function $\mathcal{L}(\theta)$ when the parameters θ are changed





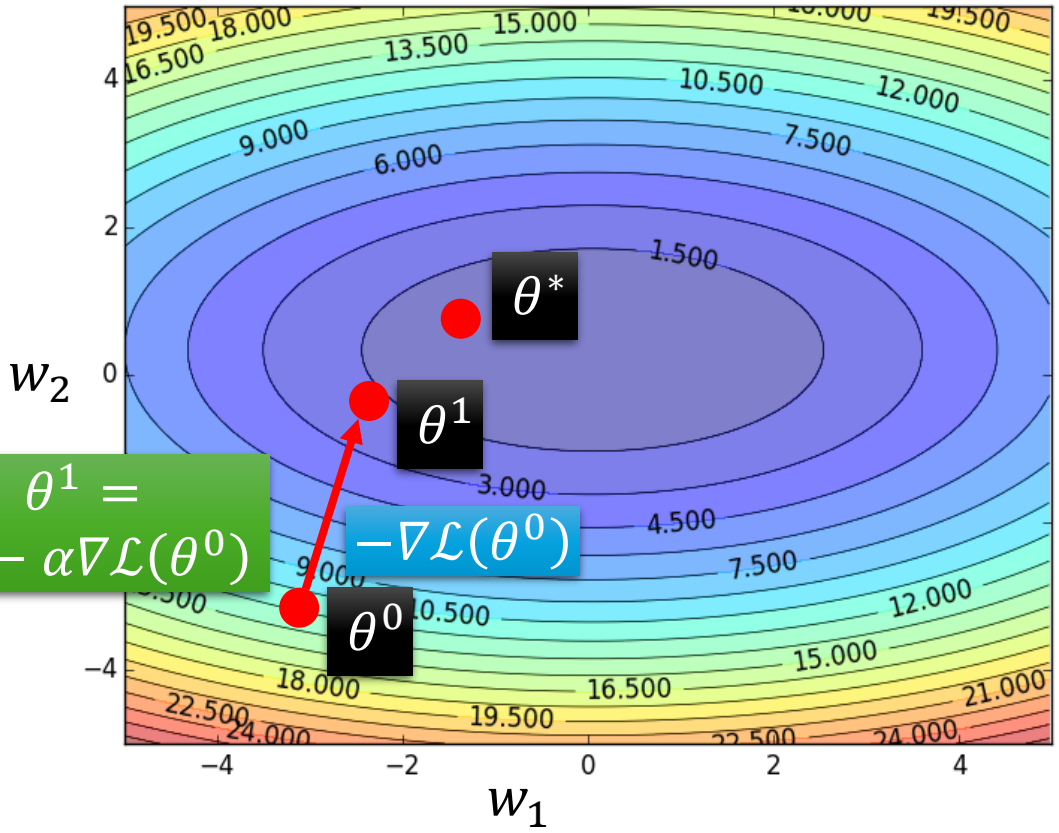
Gradient Descent Algorithm

- Steps in the ***gradient descent algorithm***:
  1. Randomly initialize the model parameters, θ^0
  2. Compute the gradient of the loss function at the initial parameters θ^0: $\nabla\mathcal{L}(\theta^0)$
  3. Update the parameters as: $\theta^{new} = \theta^0 - \alpha\nabla\mathcal{L}(\theta^0)$
     - Where α is the learning rate
  4. Go to step 2 and repeat (until a terminating criterion is reached)



Gradient Descent Algorithm

- Example: a NN with only 2 parameters w_1 and w_2, i.e., $\theta = \{w_1, w_2\}$
  - The different colors represent the values of the loss (minimum loss θ^* is ≈ 1.3)



1. Randomly pick a starting point θ^0

2. Compute the gradient at θ^0, $\nabla\mathcal{L}(\theta^0)$

3. Times the learning rate η, and update θ, $\theta^{new} = \theta^0 - \alpha\nabla\mathcal{L}(\theta^0)$

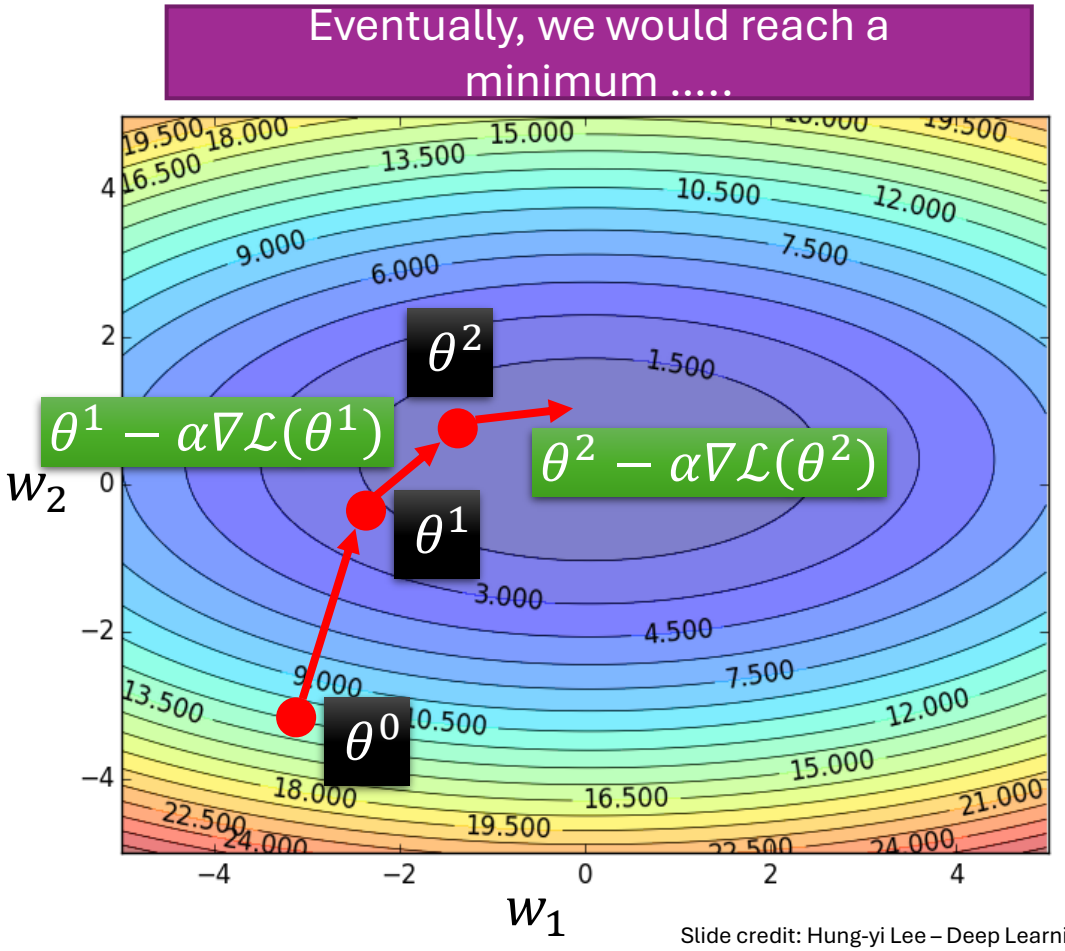
4. Go to step 2, repeat

$$\nabla\mathcal{L}(\theta^0) = \begin{bmatrix} \partial\mathcal{L}(\theta^0)/\partial w_1 \\ \partial\mathcal{L}(\theta^0)/\partial w_2 \end{bmatrix}$$

Slide credit: Hung-yi Lee – Deep Learning Tutorial

Gradient Descent Algorithm

- Example (contd.)



2. Compute the gradient at θ^{old}, $\nabla\mathcal{L}(\theta^{old})$

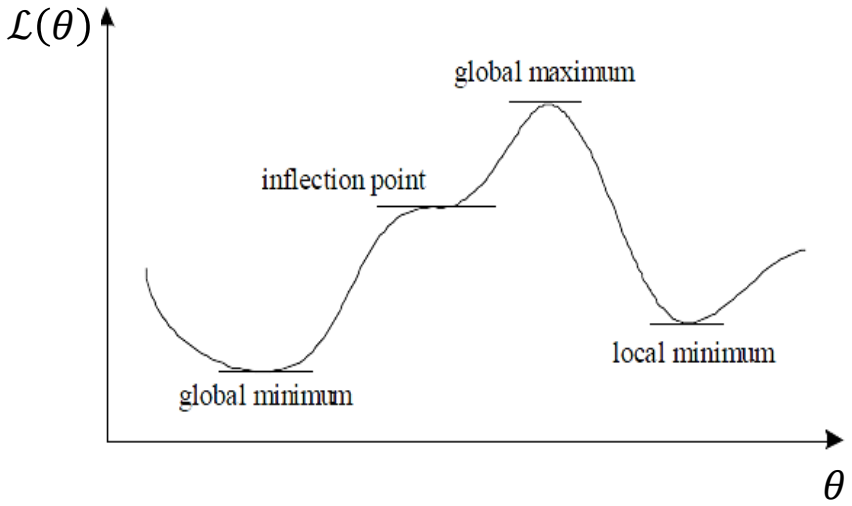
3. Times the learning rate η, and update θ, $\theta^{new} = \theta^{old} - \alpha\nabla\mathcal{L}(\theta^{old})$

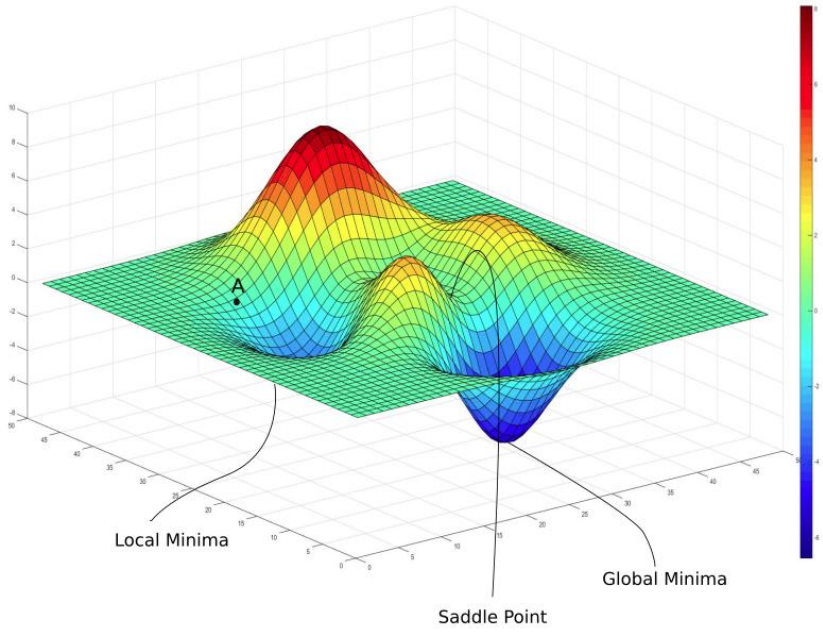
4. Go to step 2, repeat

Slide credit: Hung-yi Lee – Deep Learning Tutorial

Gradient Descent Algorithm

- Gradient descent algorithm stops when a local minimum of the loss surface is reached
  - GD does not guarantee reaching a global minimum
  - However, empirical evidence suggests that GD works well for NNs

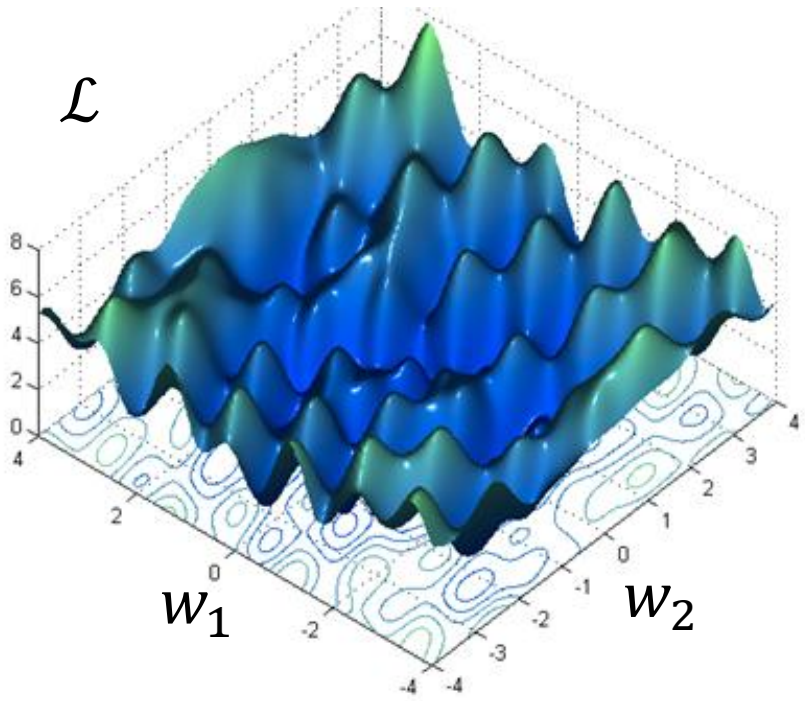




Picture from: https://blog.paperspace.com/intro-to-optimization-in-deep-learning-gradient-descent/

Gradient Descent Algorithm

- For most tasks, the loss surface $\mathcal{L}(\theta)$ is highly complex (and non-convex)
  - Random initialization in NNs results in different initial parameters θ^0 every time the NN is trained
    - Gradient descent may reach different minima at every run
    - Therefore, NN will produce different predicted outputs
  - In addition, currently we don't have algorithms that guarantee reaching a global minimum for an arbitrary loss function



Slide credit: Hung-yi Lee – Deep Learning Tutorial

Backpropagation

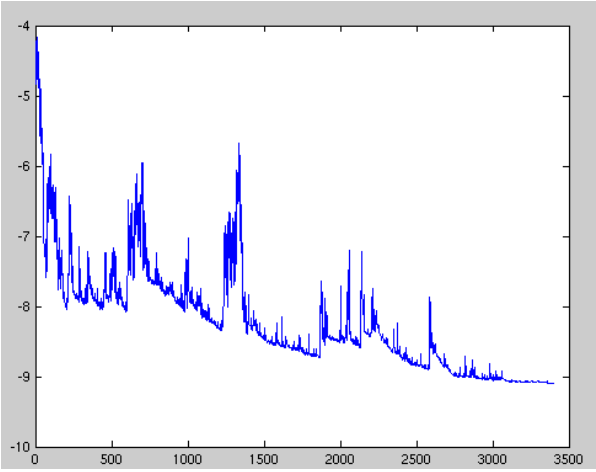
- Modern NNs employ the ***backpropagation*** method for calculating the gradients of the loss function $\nabla\mathcal{L}(\theta) = \partial\mathcal{L}/\partial\theta_i$
  - Backpropagation is short for "backward propagation"
- For training NNs, forward propagation (forward pass) refers to passing the inputs x through the hidden layers to obtain the model outputs (predictions) y
  - The loss $\mathcal{L}(y, \hat{y})$ function is then calculated
  - Backpropagation traverses the network in reverse order, from the outputs y backward toward the inputs x to calculate the gradients of the loss $\nabla\mathcal{L}(\theta)$
  - The chain rule is used for calculating the partial derivatives of the loss function with respect to the parameters θ in the different layers in the network
- Each update of the model parameters θ during training takes one forward and one backward pass (e.g., of a batch of inputs)
- Automatic calculation of the gradients (automatic differentiation) is available in all current deep learning libraries
  - It significantly simplifies the implementation of deep learning algorithms, since it obviates deriving the partial derivatives of the loss function by hand

Mini-batch Gradient Descent

- It is wasteful to compute the loss over the entire training dataset to perform a single parameter update for large datasets
  - E.g., ImageNet has 14M images
  - Therefore, GD (a.k.a. vanilla GD) is almost always replaced with mini-batch GD
- ***Mini-batch gradient descent***
  - Approach:
    - Compute the loss $\mathcal{L}(\theta)$ on a mini-batch of images, update the parameters θ, and repeat until all images are used
    - At the next epoch, shuffle the training data, and repeat the above process
  - Mini-batch GD results in much faster training
  - Typical mini-batch size: 32 to 256 images
  - It works because the gradient from a mini-batch is a good approximation of the gradient from the entire training set

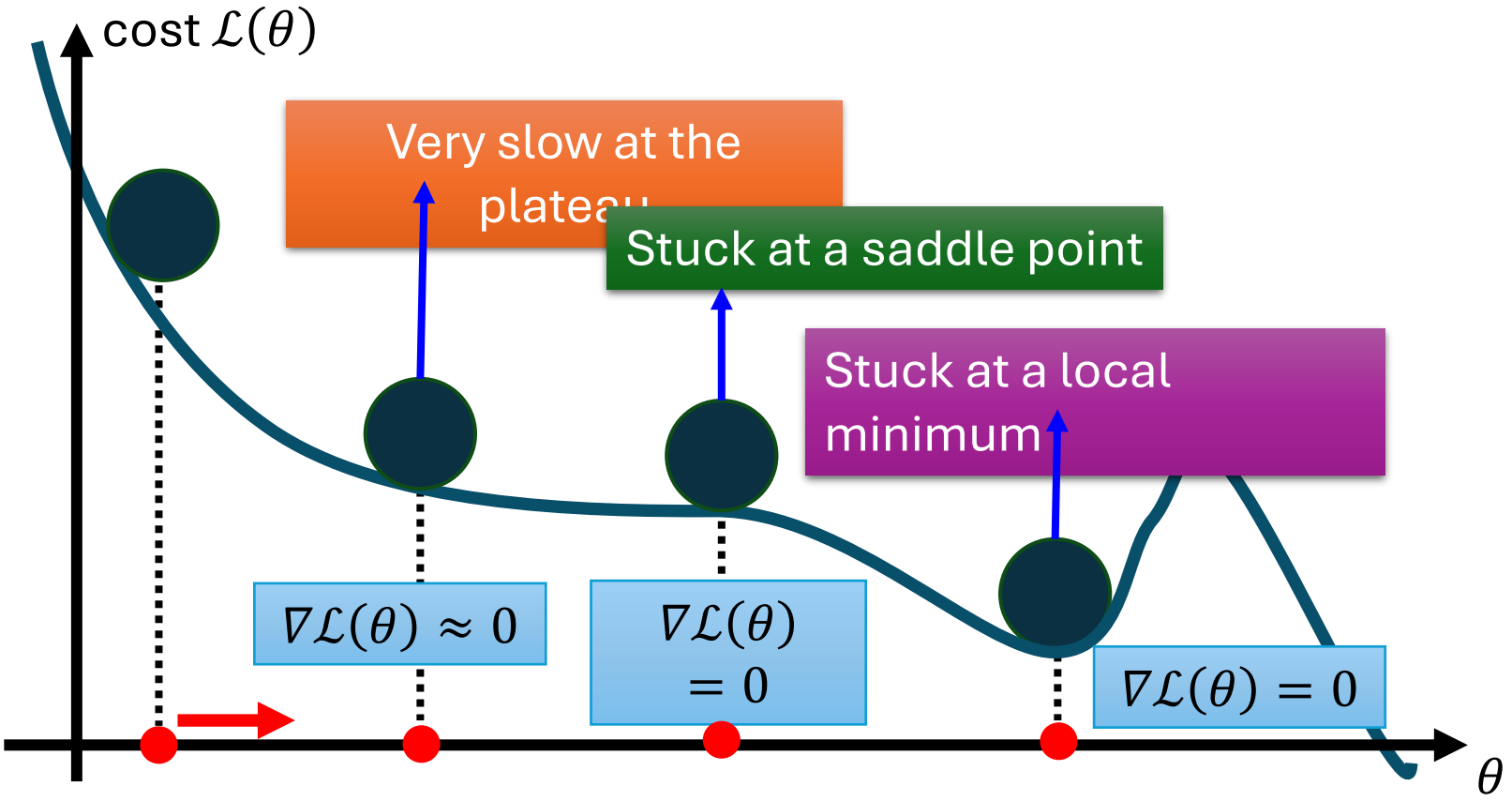
Stochastic Gradient Descent

- ***Stochastic gradient descent***
  - SGD uses mini-batches that consist of a single input example
    - E.g., one image mini-batch
  - Although this method is very fast, it may cause significant fluctuations in the loss function
    - Therefore, it is less commonly used, and mini-batch GD is preferred
  - In most DL libraries, SGD typically means a mini-batch GD (with an option to add momentum)



Problems with Gradient Descent

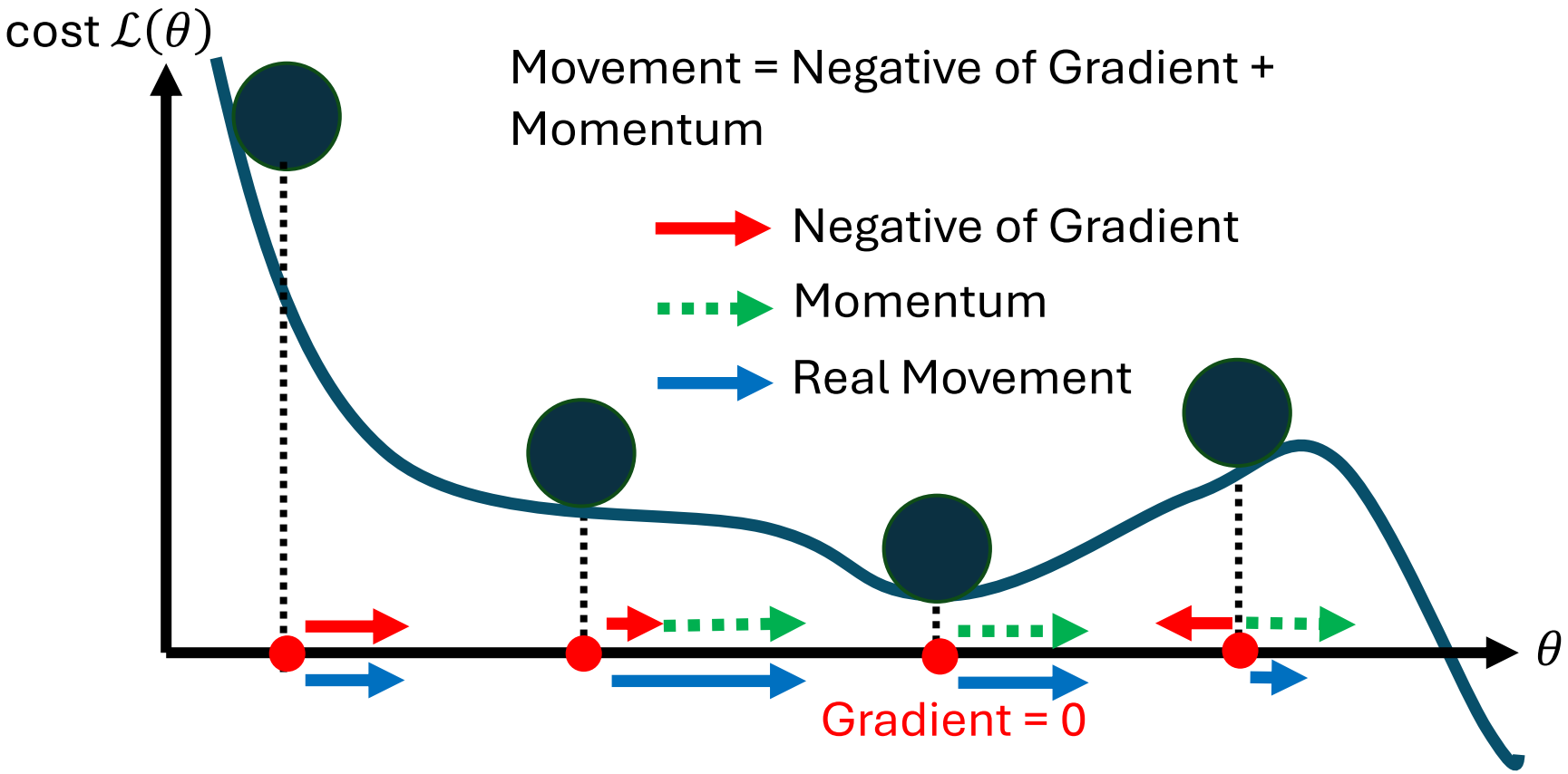
- Besides the local minima problem, the GD algorithm can be very slow at plateaus, and it can get stuck at saddle points



Slide credit: Hung-yi Lee – Deep Learning Tutorial

Gradient Descent with Momentum

- ***Gradient descent with momentum*** uses the momentum of the gradient for parameter optimization



Slide credit: Hung-yi Lee – Deep Learning Tutorial

Gradient Descent with Momentum

- Parameters update in ***GD with momentum*** at iteration t: $\theta^t = \theta^{t-1} - V^t$
  - Where: $V^t = \beta V^{t-1} + \alpha \nabla \mathcal{L}(\theta^{t-1})$
  - I.e., $\theta^t = \theta^{t-1} - \alpha \nabla \mathcal{L}(\theta^{t-1}) - \beta V^{t-1}$
- Compare to vanilla GD: $\theta^t = \theta^{t-1} - \alpha \nabla \mathcal{L}(\theta^{t-1})$
  - Where θ^{t-1} are the parameters from the previous iteration $t-1$
- The term V^t is called momentum
  - This term accumulates the gradients from the past several steps, i.e.,

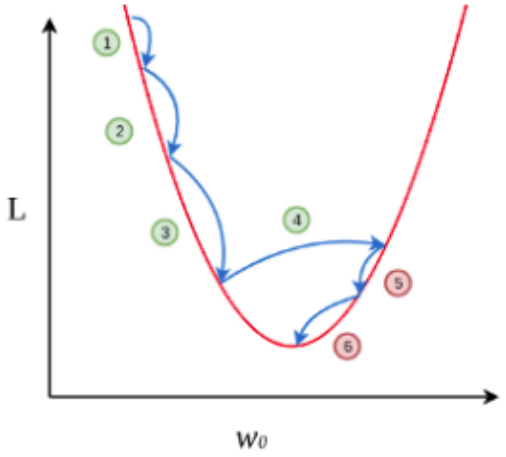
$$\begin{aligned} V^t &= \beta V^{t-1} + \alpha \nabla \mathcal{L}(\theta^{t-1}) \\ &= \beta\left(\beta V^{t-2} + \alpha \nabla \mathcal{L}(\theta^{t-2})\right) + \alpha \nabla \mathcal{L}(\theta^{t-1}) \\ &= \beta^2 V^{t-2} + \beta\alpha \nabla \mathcal{L}(\theta^{t-2}) + \alpha \nabla \mathcal{L}(\theta^{t-1}) \\ &= \beta^3 V^{t-3} + \beta^2\alpha \nabla \mathcal{L}(\theta^{t-3}) + \beta\alpha \nabla \mathcal{L}(\theta^{t-2}) + \alpha \nabla \mathcal{L}(\theta^{t-1}) \end{aligned}$$

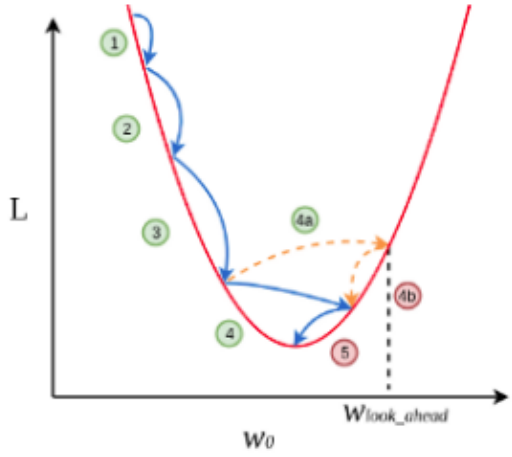
  - This term is analogous to a momentum of a heavy ball rolling down the hill
- The parameter β is referred to as a coefficient of momentum
  - A typical value of the parameter β is 0.9
- This method updates the parameters θ in the direction of the weighted average of the past gradients

Nesterov Accelerated Momentum

- ***Gradient descent with Nesterov accelerated momentum***
  - Parameter update: $\theta^t = \theta^{t-1} - V^t$
    - Where: $V^t = \beta V^{t-1} + \alpha \nabla \mathcal{L}(\theta^{t-1} + \beta V^{t-1})$
  - The term $\theta^{t-1} + \beta V^{t-1}$ allows to predict the position of the parameters in the next step (i.e., $\theta^t \approx \theta^{t-1} + \beta V^{t-1}$)
  - The gradient is calculated with respect to the approximate future position of the parameters in the next iteration, θ^t, calculated at iteration $t-1$

GD with momentum





GD with Nesterov momentum

Picture from: https://towardsdatascience.com/learning-parameters-part-2-a190bef2d12

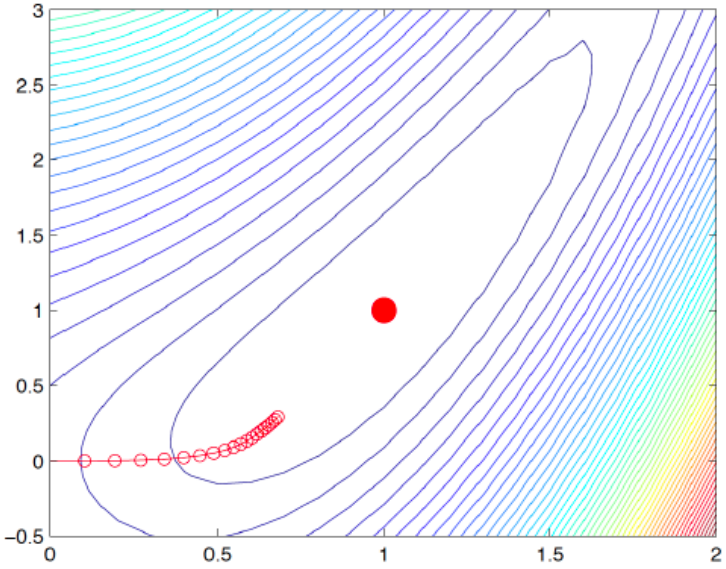
Adam

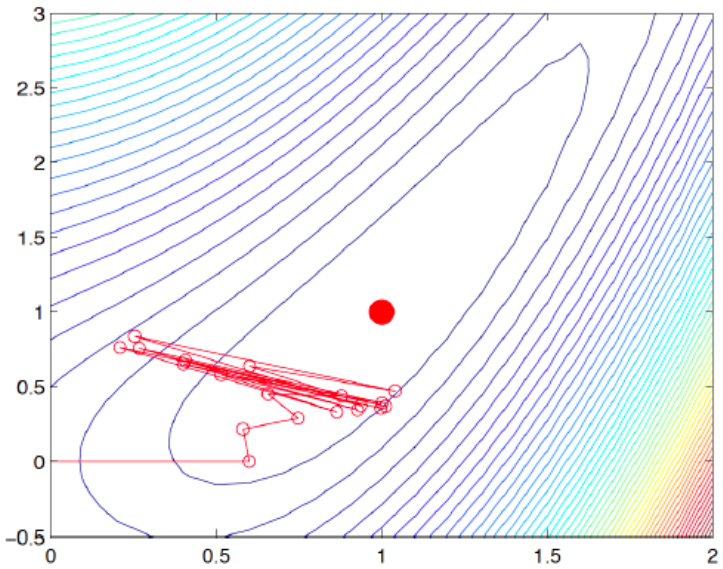
- ***Adaptive Moment Estimation (Adam)***
  - Adam combines insights from the momentum optimizers that accumulate the values of past gradients, and it also introduces new terms based on the second moment of the gradient
    - Similar to GD with momentum, Adam computes a **weighted average of past gradients** (first moment of the gradient), i.e., $V^t = \beta_1 V^{t-1} + (1-\beta_1)\nabla\mathcal{L}(\theta^{t-1})$
    - Adam also computes a **weighted average of past squared gradients** (second moment of the gradient), , i.e., $U^t = \beta_2 U^{t-1} + (1-\beta_2)\left(\nabla\mathcal{L}(\theta^{t-1})\right)^2$
  - The parameter update is: $\theta^t = \theta^{t-1} - \alpha \frac{\hat{V}^t}{\sqrt{\hat{U}^t}+\epsilon}$
    - Where: $\hat{V}^t = \frac{V^t}{1-\beta_1}$ and $\hat{U}^t = \frac{U^t}{1-\beta_2}$
    - The proposed default values are $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 10^{-8}$
- Other commonly used optimization methods include:
  - Adagrad, Adadelta, RMSprop, Nadam, etc.
  - Most commonly used optimizers nowadays are Adam and SGD with momentum

Learning Rate

- ***Learning rate***
  - The gradient tells us the direction in which the loss has the steepest rate of increase, but it does not tell us how far along the opposite direction we should step
  - Choosing the learning rate (also called the step size) is one of the most important hyper-parameter settings for NN training

LR too small

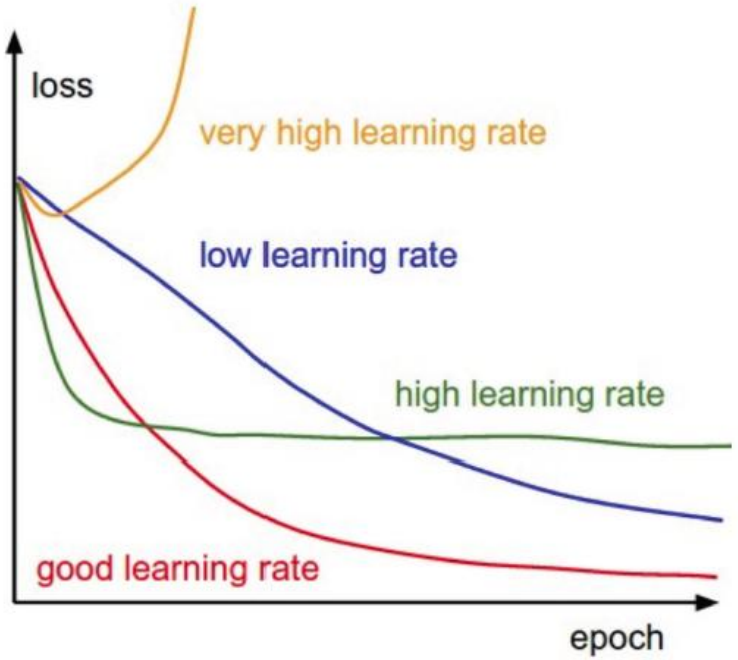




LR too large

Learning Rate

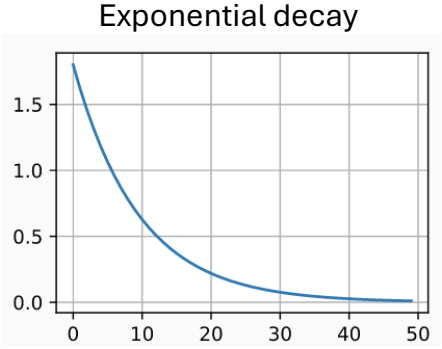
- Training loss for different learning rates
  - High learning rate: the loss increases or plateaus too quickly
  - Low learning rate: the loss decreases too slowly (takes many epochs to reach a solution)

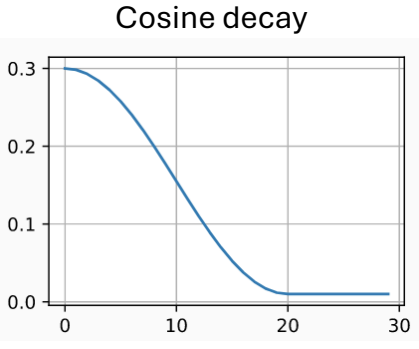


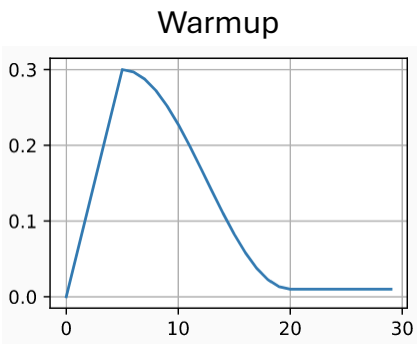
Picture from: https://cs231n.github.io/neural-networks-3/

Learning Rate Scheduling

- ***Learning rate scheduling*** is applied to change the values of the learning rate during the training
  - ***Annealing*** is reducing the learning rate over time (a.k.a. learning rate decay)
    - Approach 1: reduce the learning rate by some factor every few epochs
      - Typical values: reduce the learning rate by a half every 5 epochs, or divide by 10 every 20 epochs
    - Approach 2: exponential or cosine decay gradually reduce the learning rate over time
    - Approach 3: reduce the learning rate by a constant (e.g., by half) whenever the validation loss stops improving
      - In TensorFlow: tf.keras.callbacks.ReduceLROnPleateau()
        - Monitor: validation loss, factor: 0.1 (i.e., divide by 10), patience: 10 (how many epochs to wait before applying it), Minimum learning rate: 1e-6 (when to stop)
  - ***Warmup*** is gradually increasing the learning rate initially, and afterward let it cool down until the end of the training

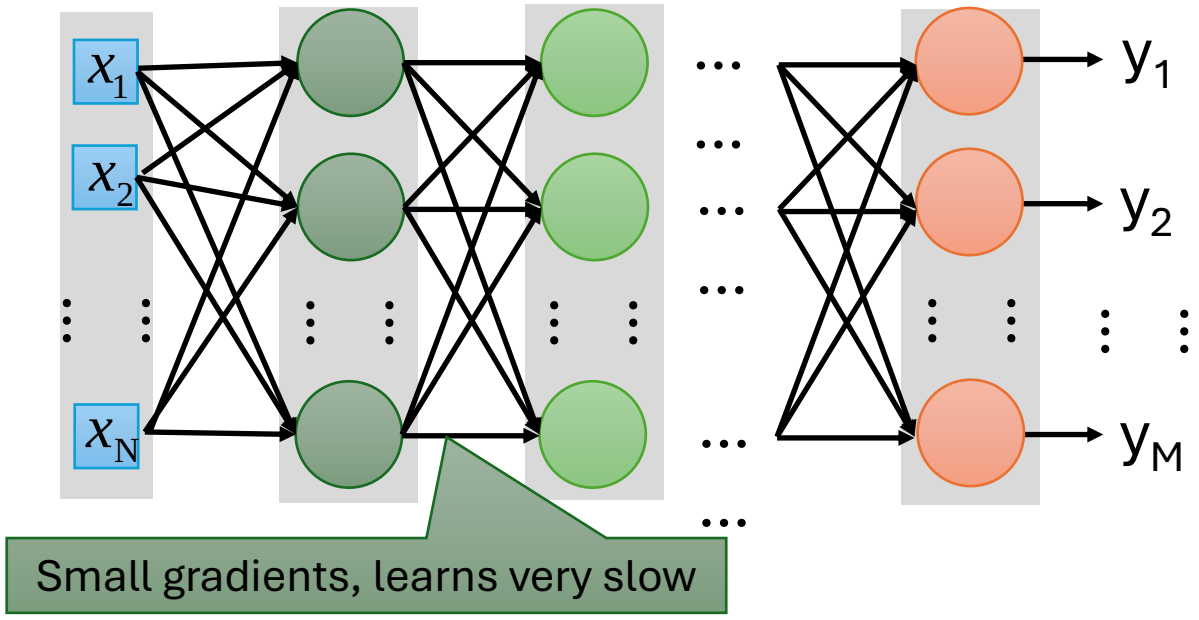






Vanishing Gradient Problem

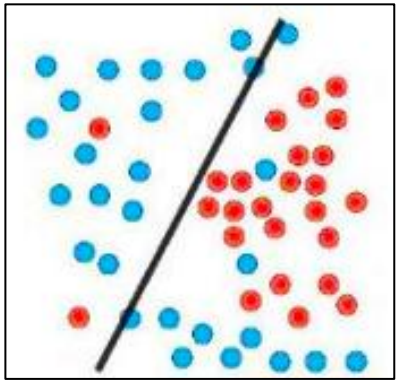
- In some cases, during training, the gradients can become either very small (vanishing gradients) of very large (exploding gradients)
  - They result in very small or very large update of the parameters
  - Solutions: change learning rate, ReLU activations, regularization, LSTM units in RNNs



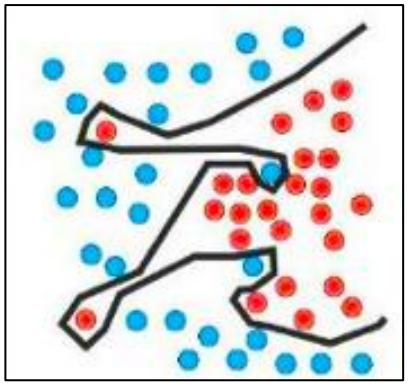
Slide credit: Hung-yi Lee – Deep Learning Tutorial

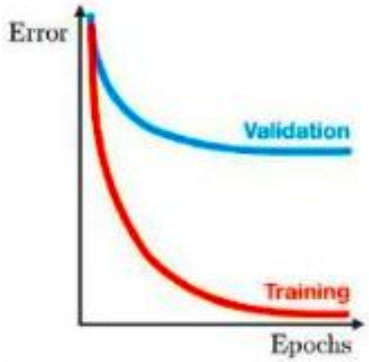
Generalization

- ***Underfitting***
  - The model is too "simple" to represent all the relevant class characteristics
  - E.g., model with too few parameters
  - Produces high error on the training set and high error on the validation set



- ***Overfitting***
  - The model is too "complex" and fits irrelevant characteristics (noise) in the data
  - E.g., model with too many parameters
  - Produces low error on the training error and high error on the validation set

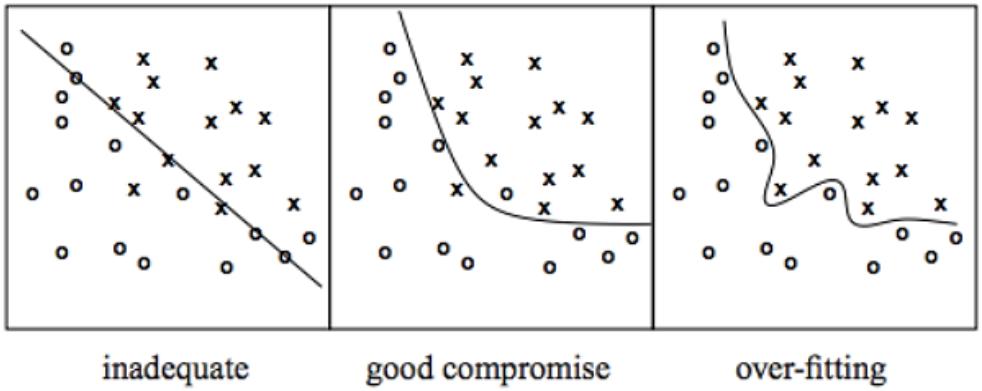




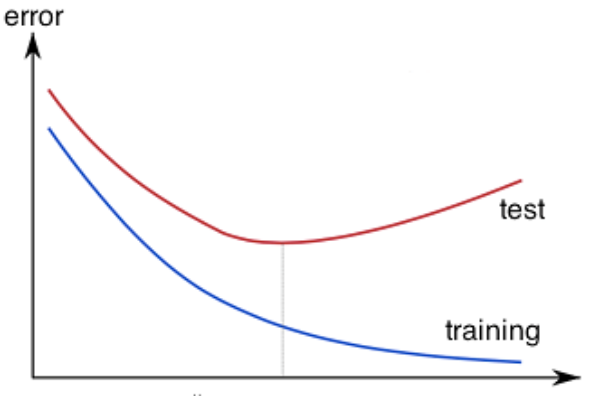
Overfitting

Generalization

- Overfitting – a model with high capacity fits the noise in the data instead of the underlying relationship



- The model may fit the training data very well, but fails to generalize to new examples (test or validation data)



Picture from: http://cs231n.github.io/assets/nn1/layer_sizes.jpeg

Regularization: Weight Decay

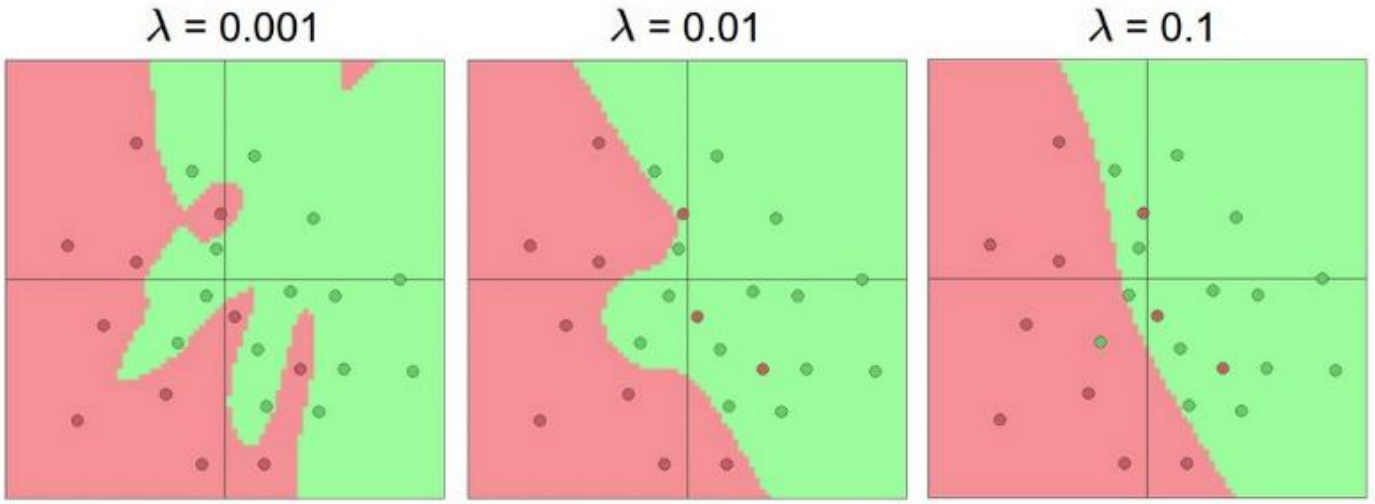
- ***ℓ_2 weight decay***
  - A regularization term that penalizes large weights is added to the loss function

$$\mathcal{L}_{reg}(\theta) = \underbrace{\mathcal{L}(\theta)}_{\text{Data loss}} + \underbrace{\lambda \sum_k \theta_k^2}_{\text{Regularization loss}}$$

  - For every weight in the network, we add the regularization term to the loss value
    - During gradient descent parameter update, every weight is decayed linearly toward zero
  - The weight decay coefficient λ determines how dominant the regularization is during the gradient computation

Regularization: Weight Decay

- Effect of the decay coefficient λ
  - Large weight decay coefficient → penalty for weights with large values



Regularization: Weight Decay

- ***ℓ_1 weight decay***
  - The regularization term is based on the ℓ_1 norm of the weights

$$\mathcal{L}_{reg}(\theta) = \mathcal{L}(\theta) + \lambda \sum_k |\theta_k|$$

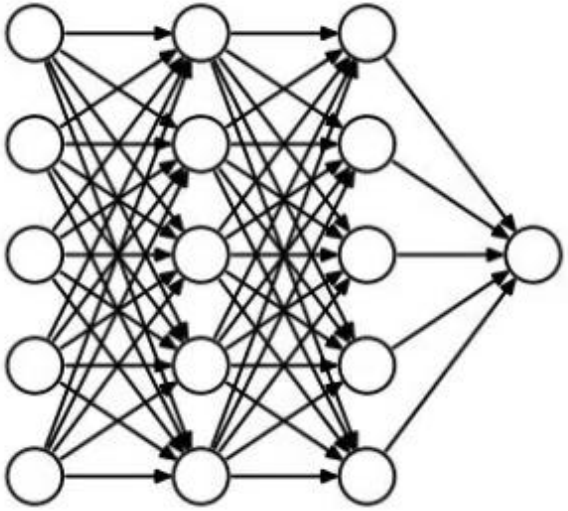
  - ℓ_1 weight decay is less common with NN
    - Often performs worse than ℓ_2 weight decay
  - It is also possible to combine ℓ_1 and ℓ_2 regularization
    - Called elastic net regularization

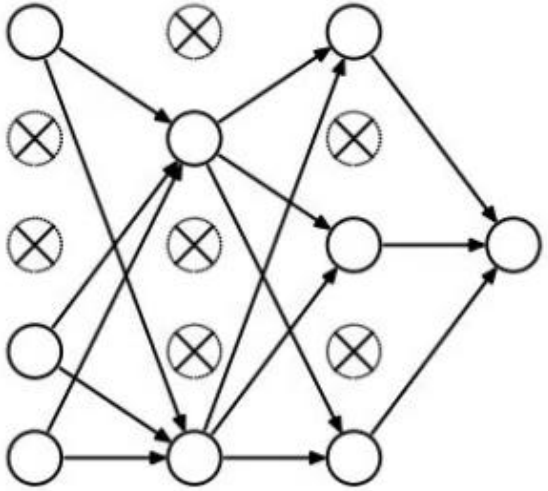
$$\mathcal{L}_{reg}(\theta) = \mathcal{L}(\theta) + \lambda_1 \sum_k |\theta_k| + \lambda_2 \sum_k \theta_k^2$$

Regularization: Dropout

Regularization

- ***Dropout***
  - Randomly drop units (along with their connections) during training
  - Each unit is retained with a fixed dropout rate p, independent of other units
  - The hyper-parameter p needs to be chosen (tuned)
    - Often, between 20% and 50% of the units are dropped



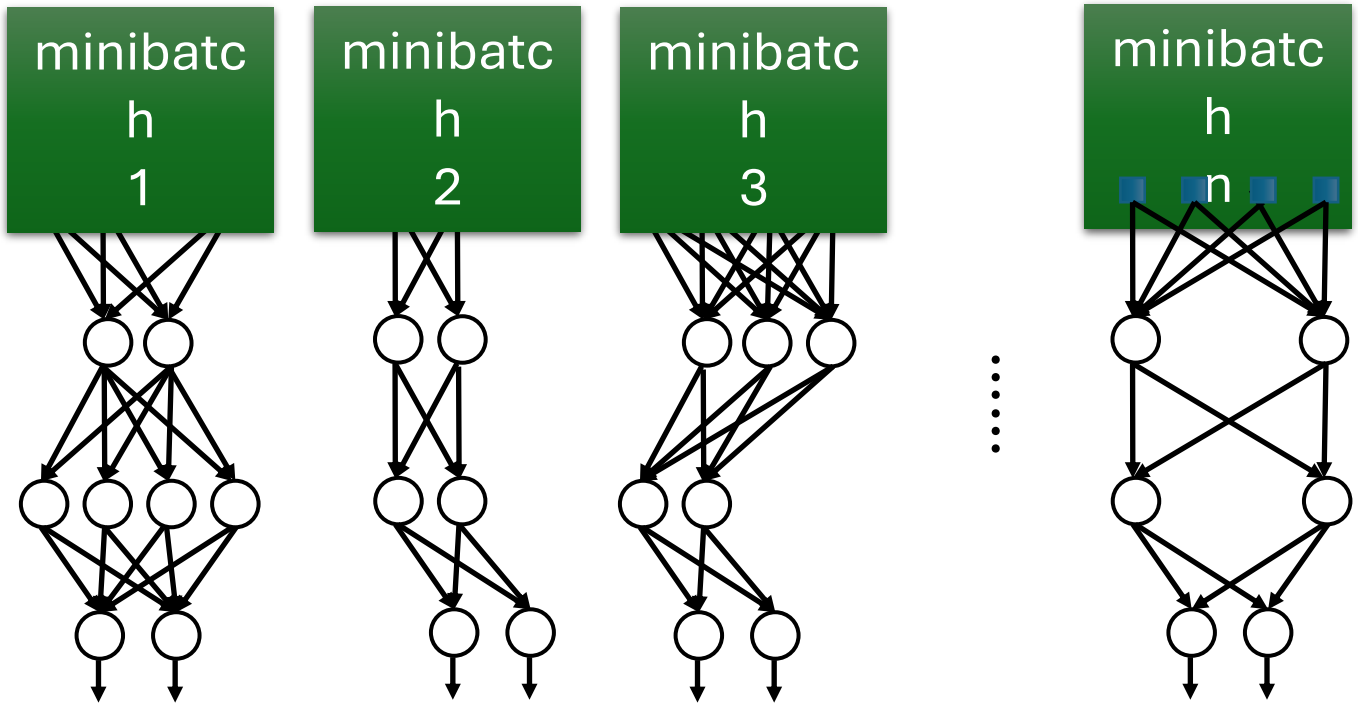


Slide credit: Hung-yi Lee – Deep Learning Tutorial

Regularization: Dropout

Regularization

- Dropout is a kind of ensemble learning
  - Using one mini-batch to train one network with a slightly different architecture

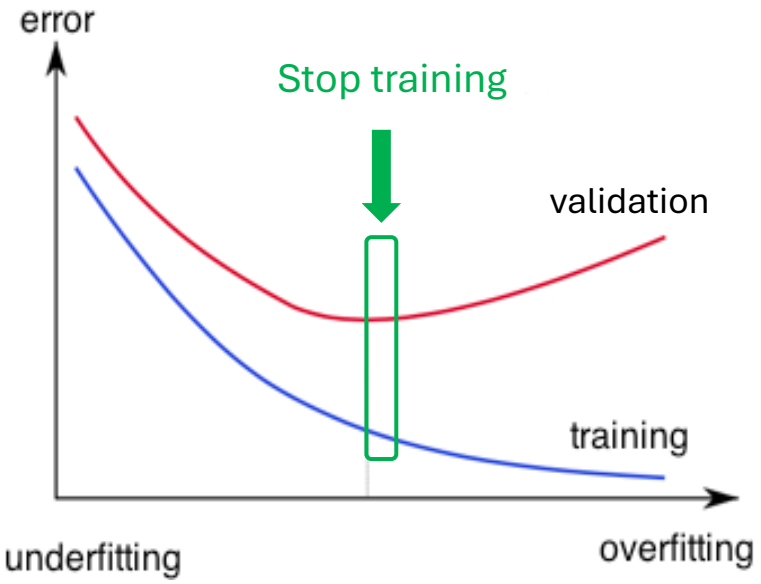


Slide credit: Hung-yi Lee – Deep Learning Tutorial

Regularization: Early Stopping

Regularization

- ***Early-stopping***
  - During model training, use a validation set
    - E.g., validation/train ratio of about 25% to 75%
  - Stop when the validation accuracy (or loss) has not improved after n epochs
    - The parameter n is called patience



Batch Normalization

Regularization

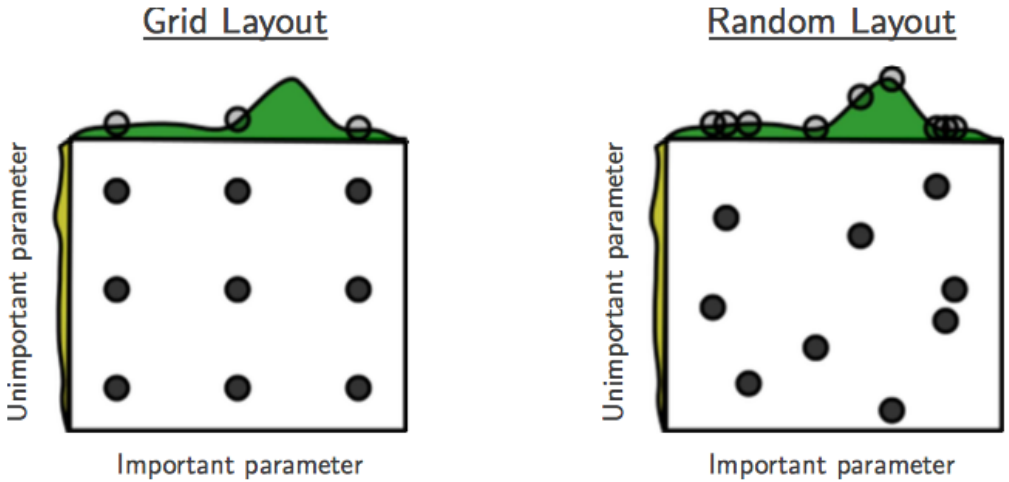
- ***Batch normalization layers*** act similar to the data preprocessing steps mentioned earlier
  - They calculate the mean μ and variance σ of a batch of input data, and normalize the data x to a zero mean and unit variance
  - I.e., $\hat{x} = \frac{x-\mu}{\sigma}$
- BatchNorm layers alleviate the problems of proper initialization of the parameters and hyper-parameters
  - Result in faster convergence training, allow larger learning rates
  - Reduce the internal covariate shift
- BatchNorm layers are inserted immediately after convolutional layers or fully-connected layers, and before activation layers
  - They are very common with convolutional NNs

Hyper-parameter Tuning

- Training NNs can involve setting many ***hyper-parameters***
- The most common hyper-parameters include:
  - Number of layers, and number of neurons per layer
  - Initial learning rate
  - Learning rate decay schedule (e.g., decay constant)
  - Optimizer type
- Other hyper-parameters may include:
  - Regularization parameters (ℓ_2 penalty, dropout rate)
  - Batch size
  - Activation functions
  - Loss function
- Hyper-parameter tuning can be time-consuming for larger NNs

Hyper-parameter Tuning

- Grid search
  - Check all values in a range with a step value
- Random search
  - Randomly sample values for the parameter
  - Often preferred to grid search
- Bayesian hyper-parameter optimization
  - Is an active area of research

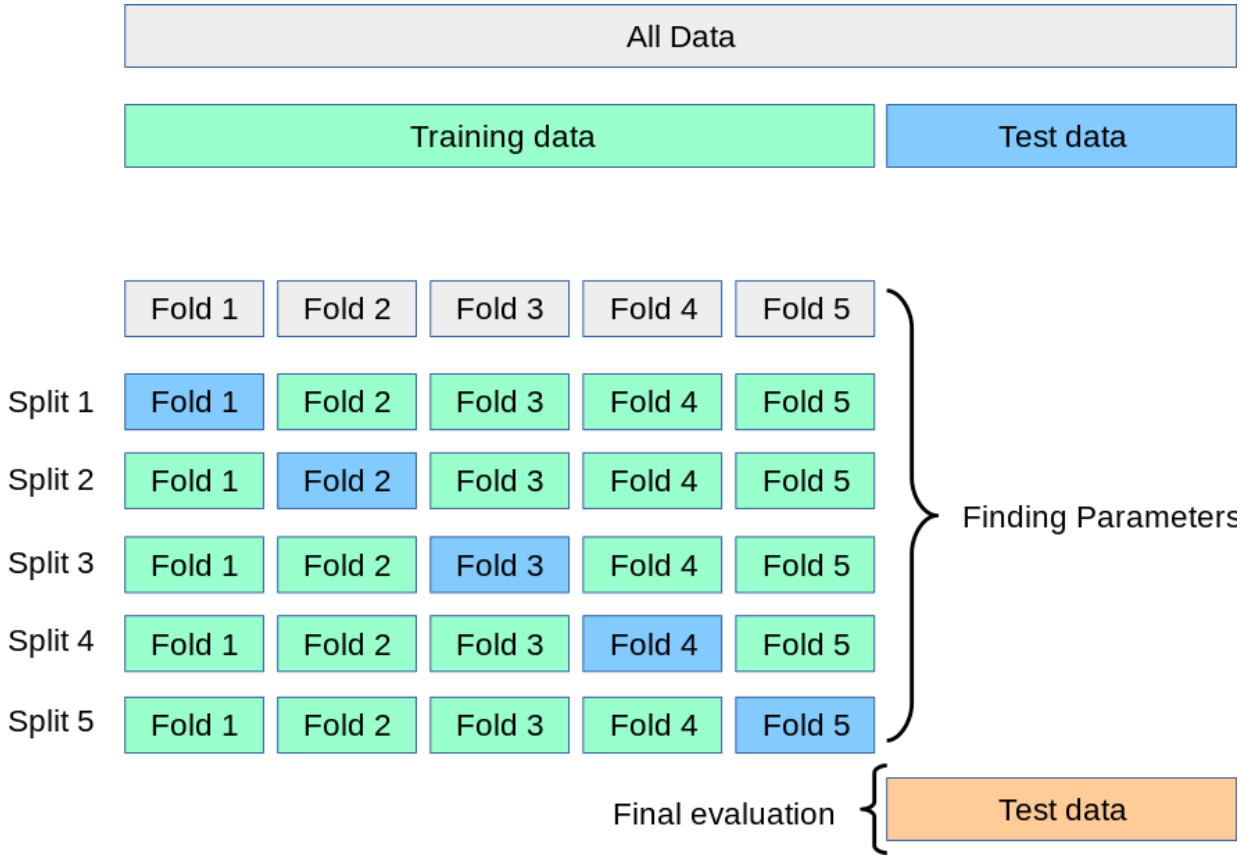


k-Fold Cross-Validation

- Using ***k-fold cross-validation*** for hyper-parameter tuning is common when the size of the training data is small
  - It also leads to a better and less noisy estimate of the model performance by averaging the results across several folds
- E.g., 5-fold cross-validation (see the figure on the next slide)
  1. Split the train data into 5 equal folds
  2. First use folds 2-5 for training and fold 1 for validation
  3. Repeat by using fold 2 for validation, then fold 3, fold 4, and fold 5
  4. Average the results over the 5 runs (for reporting purposes)
  5. Once the best hyper-parameters are determined, evaluate the model on the test data

k-Fold Cross-Validation

- Illustration of a 5-fold cross-validation



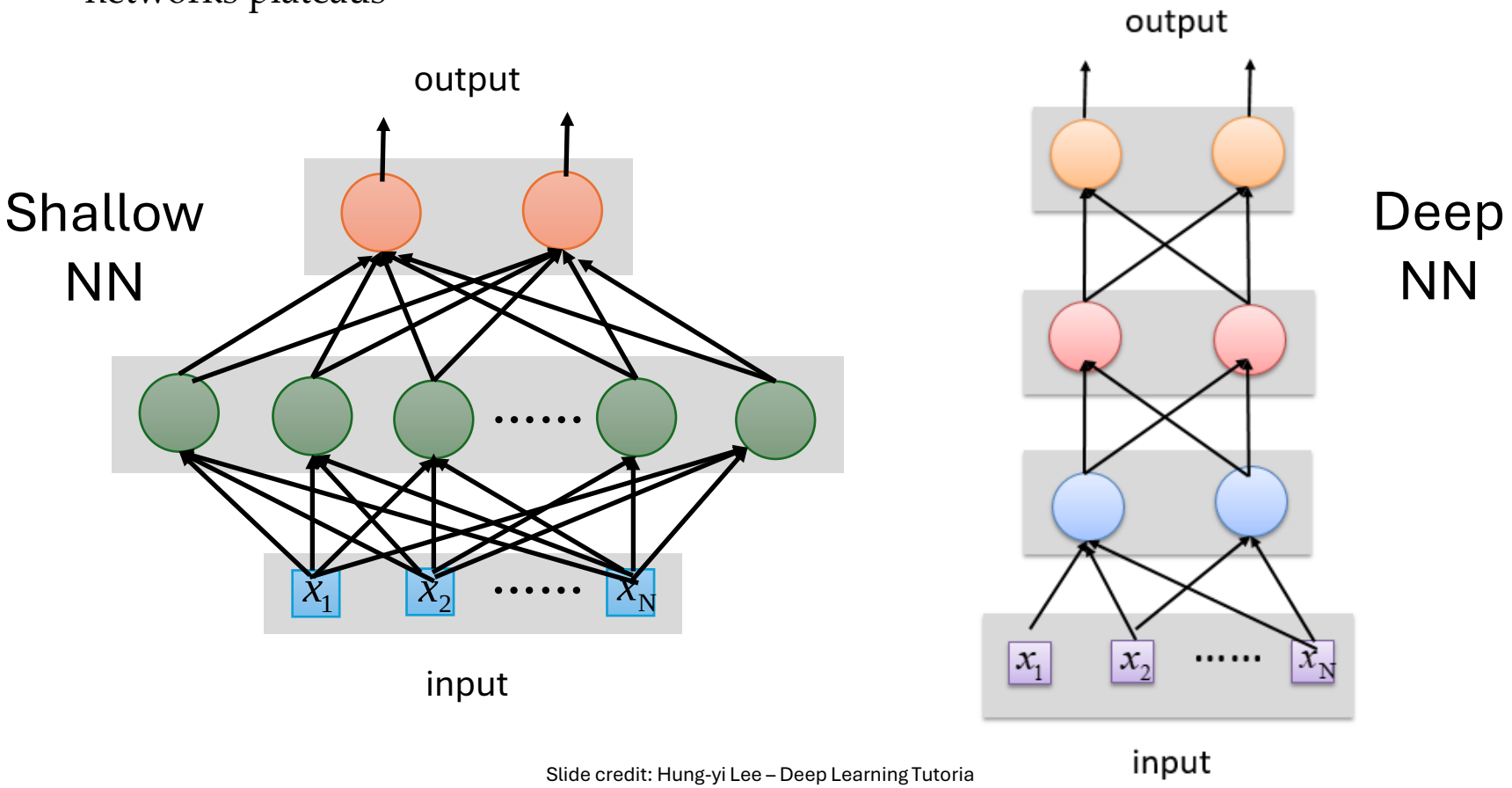
Picture from: https://scikit-learn.org/stable/modules/cross_validation.html

Ensemble Learning

- ***Ensemble learning*** is training multiple classifiers separately and combining their predictions
  - Ensemble learning often outperforms individual classifiers
  - Better results obtained with higher model variety in the ensemble
  - ***Bagging*** (**b**ootstrap **ag**gregating)
    - Randomly draw subsets from the training set (i.e., bootstrap samples)
    - Train separate classifiers on each subset of the training set
    - Perform classification based on the average vote of all classifiers
  - ***Boosting***
    - Train a classifier, and apply weights on the training set (apply higher weights on misclassified examples, focus on "hard examples")
    - Train new classifier, reweight training set according to prediction error
    - Repeat
    - Perform classification based on weighted vote of the classifiers

Deep vs Shallow Networks

- Deeper networks perform better than shallow networks
  - But only up to some limit: after a certain number of layers, the performance of deeper networks plateaus

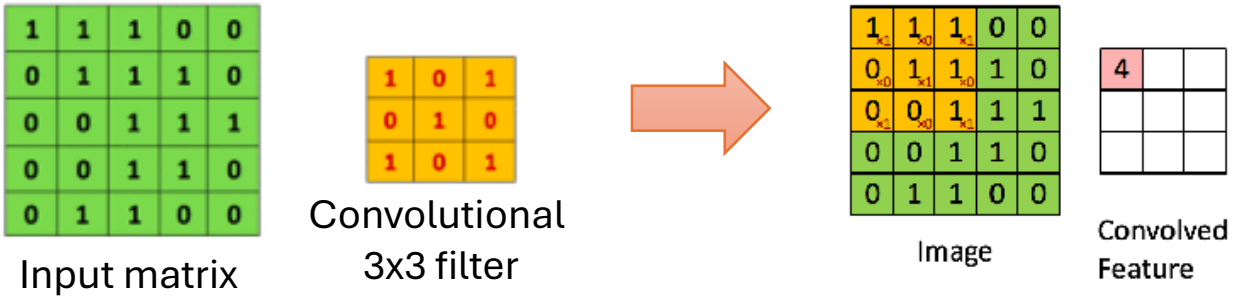


Slide credit: Hung-yi Lee – Deep Learning Tutoria

Convolutional Neural Networks (CNNs)

Convolutional Neural Networks

- ***Convolutional neural networks*** (CNNs) were primarily designed for image data
- CNNs use a convolutional operator for extracting data features
  - Allows parameter sharing
  - Efficient to train
  - Have less parameters than NNs with fully-connected layers
- CNNs are robust to spatial translations of objects in images
- A convolutional filter slides (i.e., convolves) across the image



Picture from: http://deeplearning.stanford.edu/wiki/index.php/Feature_extraction_using_convolution

Convolutional Neural Networks (CNNs)

- When the convolutional filters are scanned over the image, they capture useful features
  - E.g., edge detection by convolutions

Filter 

$$\begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & \\ 1 & & \\ 0 & 1 & 0 \end{pmatrix}$$

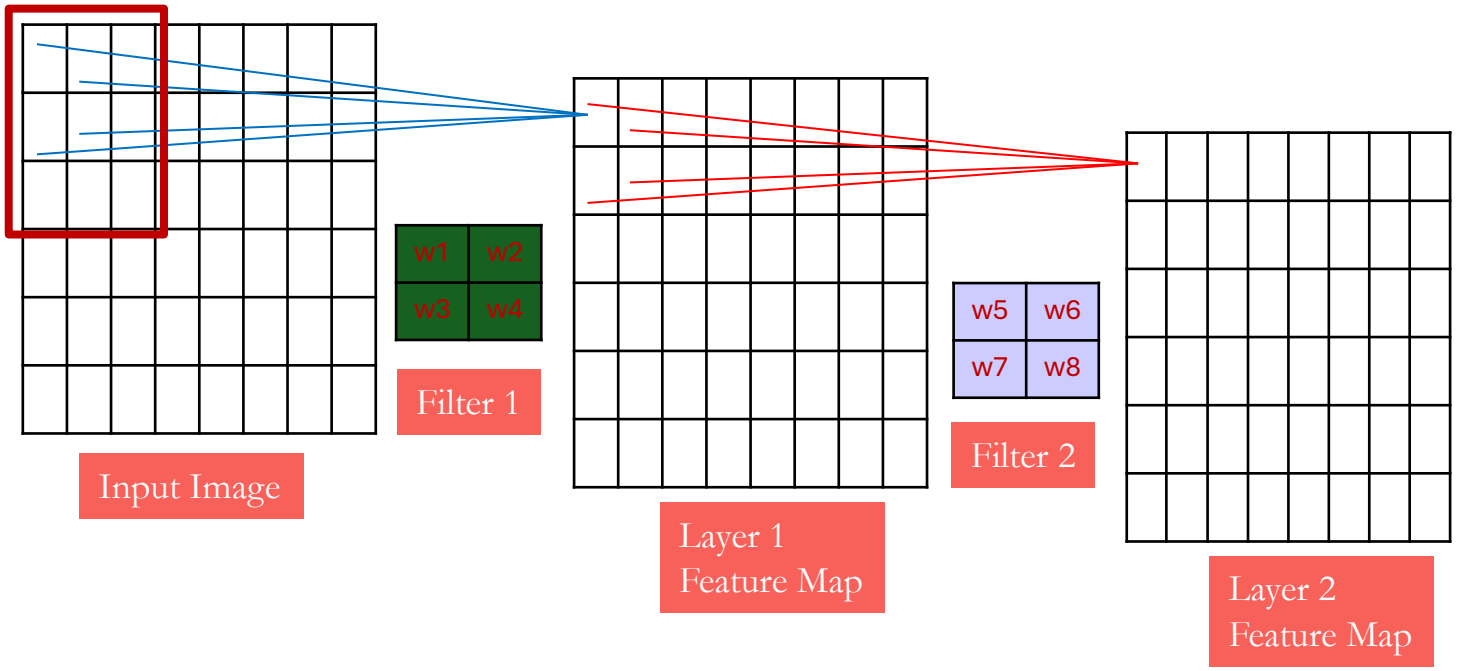
Input Image

Convoluted Image

Slide credit: Param Vir Singh – Deep Learning

Convolutional Neural Networks (CNNs)

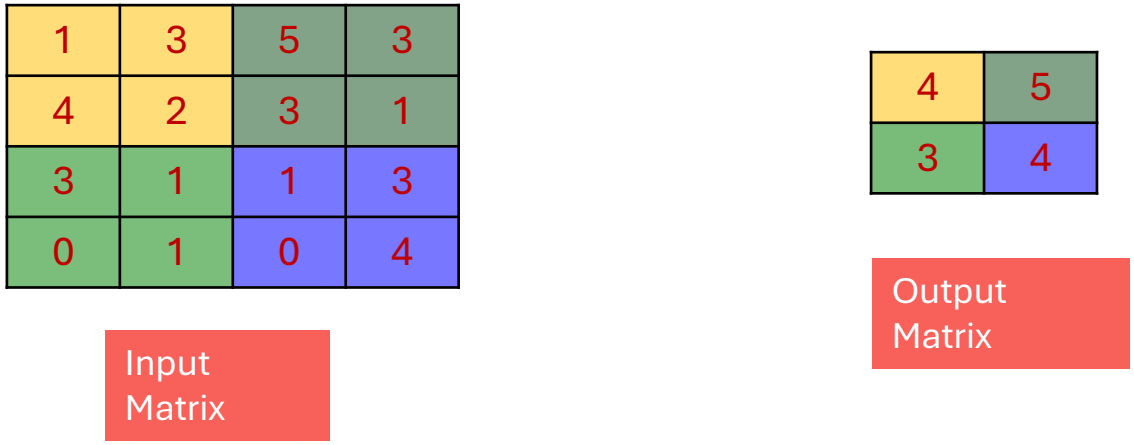
- In CNNs, hidden units in a layer are only connected to a small region of the layer before it (called local receptive field)
  - The depth of each feature map corresponds to the number of convolutional filters used at each layer



Slide credit: Param Vir Singh – Deep Learning

Convolutional Neural Networks (CNNs)

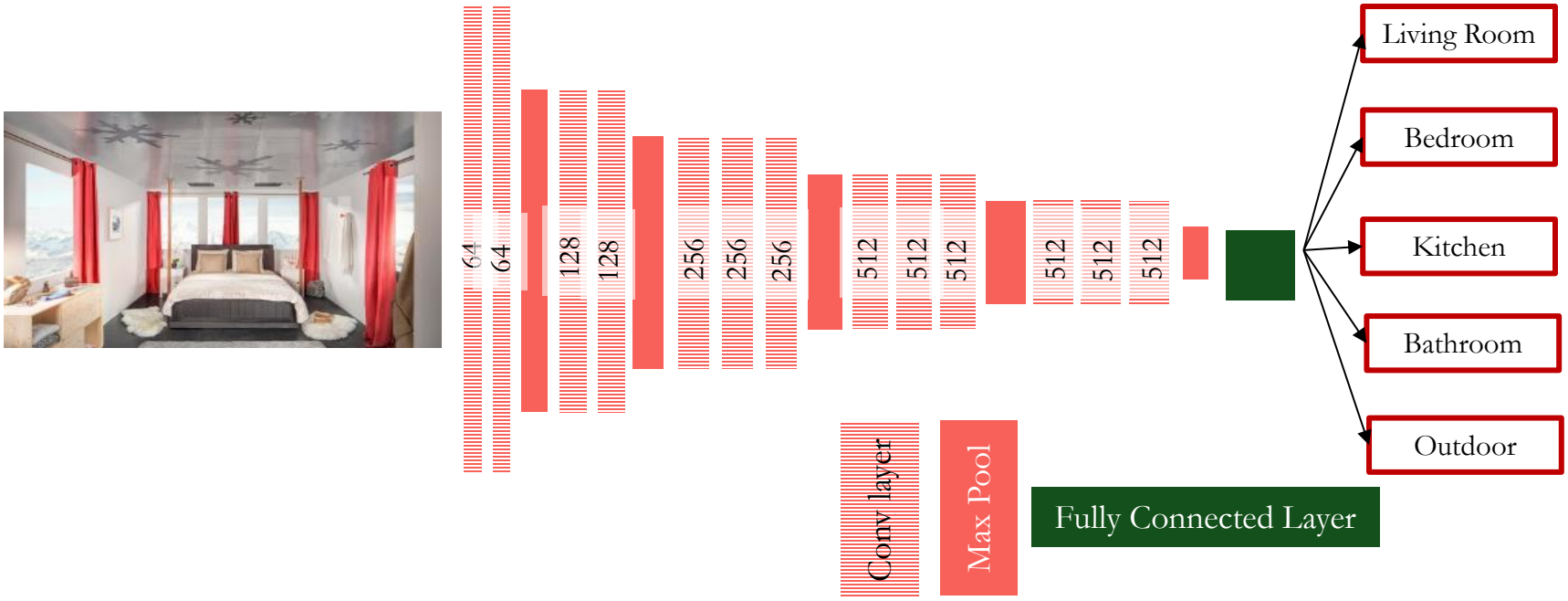
- ***Max pooling***: reports the maximum output within a rectangular neighborhood
- ***Average pooling***: reports the average output of a rectangular neighborhood
- Pooling layers reduce the spatial size of the feature maps
  - Reduce the number of parameters, prevent overfitting



Convolutional Neural Networks (CNNs)

Convolutional Neural Networks

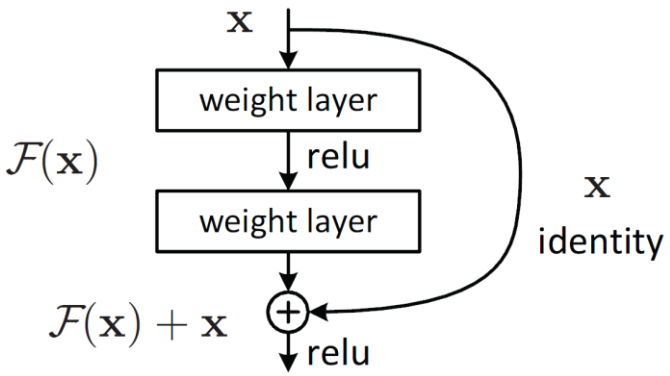
- Feature extraction architecture
  - After 2 convolutional layers, a max-pooling layer reduces the size of the feature maps (typically by 2)
  - A fully convolutional and a softmax layers are added last to perform classification

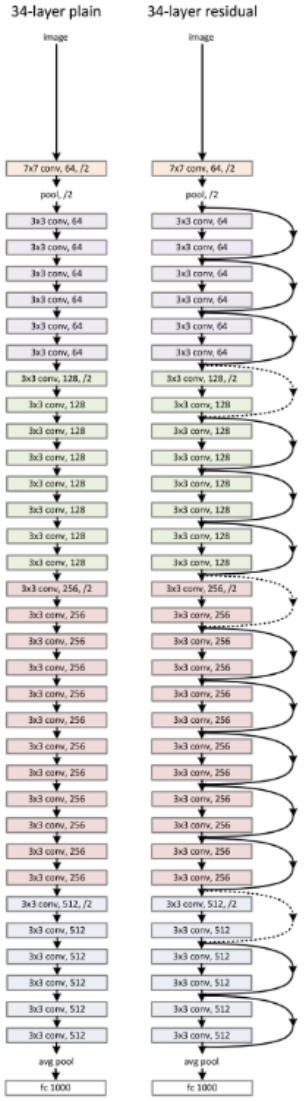


Slide credit: Param Vir Singh – Deep Learning

Residual CNNs

- ***Residual networks*** (ResNets)
  - Introduce "identity" skip connections
    - Layer inputs are propagated and added to the layer output
    - Mitigate the problem of vanishing gradients during training
    - Allow training very deep NN (with over 1,000 layers)
  - Several ResNet variants exist: 18, 34, 50, 101, 152, and 200 layers
  - Are used as base models of other state-of-the-art NNs
    - Other similar models: ResNeXT, DenseNet



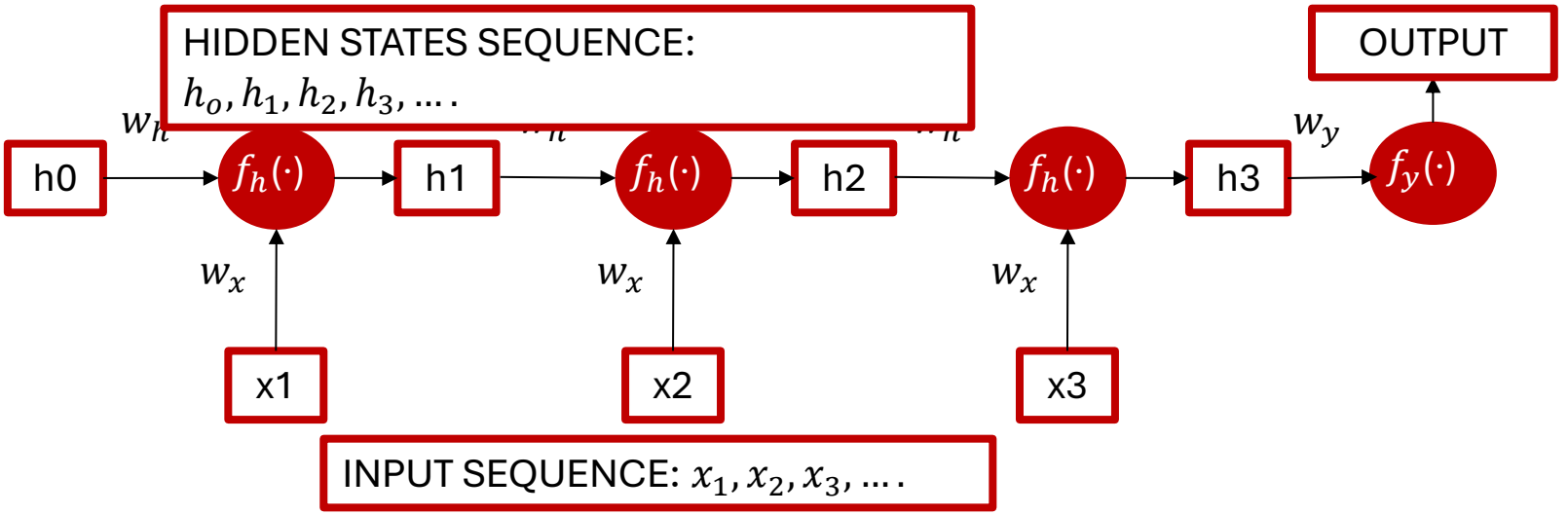


Recurrent Neural Networks (RNNs)

- ***Recurrent NNs*** are used for modeling sequential data and data with varying length of inputs and outputs
  - Videos, text, speech, DNA sequences, human skeletal data
- RNNs introduce recurrent connections between the neurons
  - This allows processing sequential data one element at a time by selectively passing information across a sequence
  - Memory of the previous inputs is stored in the model's internal state and affect the model predictions
  - Can capture correlations in sequential data
- RNNs use backpropagation-through-time for training
- RNNs are more sensitive to the vanishing gradient problem than CNNs

Recurrent Neural Networks (RNNs)

- RNN use same set of weights w_h and w_x across all time steps
  - A sequence of hidden states $\{h_o, h_o h_2, h_3, \dots\}$ is learned, which represents the memory of the network
  - The hidden state at step t, $h(t)$, is calculated based on the previous hidden state $h(t-1)$ and the input at the current step $x(t)$, i.e., $h(t) = f_h\big(w_h * h(t-1) + w_x * x(t)\big)$
  - The function $f_h(\cdot)$ is a nonlinear activation function, e.g., ReLU or tanh
- RNN shown rolled over time

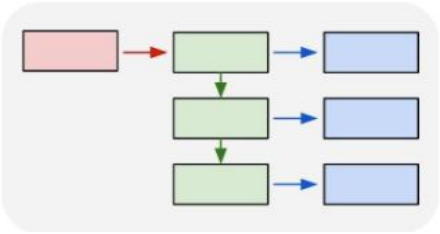
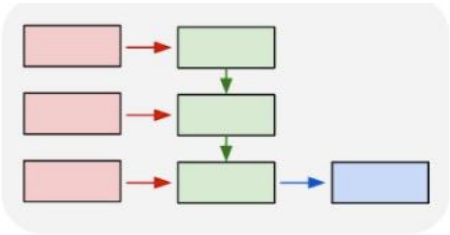
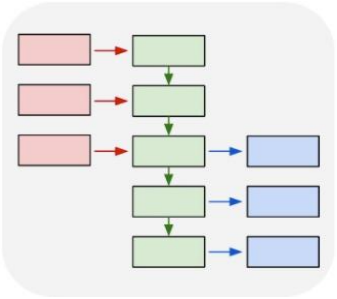


Slide credit: Param Vir Singh – Deep Learning

Recurrent Neural Networks (RNNs)

Recurrent Neural Networks

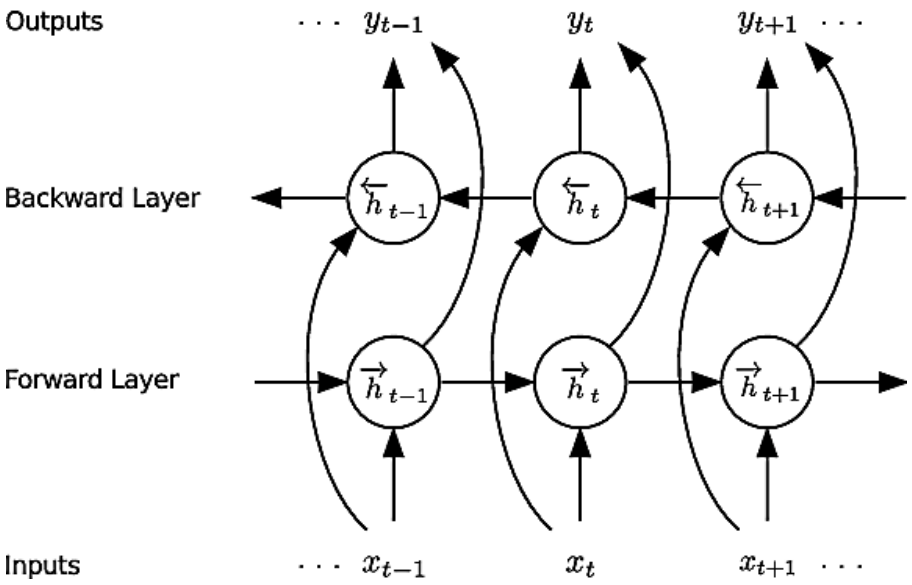
- RNNs can have one of many inputs and one of many outputs

RNN	Application	Input	Output
 one to many	Image Captioning		A person riding a motorbike on dirt road
 many to one	Sentiment Analysis	Awesome movie. Highly recommended.	Positive
 many to many	Machine Translation	Happy Diwali	शुभ दीपावली

Slide credit: Param Vir SIngh– Deep Learning

Bidirectional RNNs

- ***Bidirectional RNNs*** incorporate both forward and backward passes through sequential data
  - The output may not only depend on the previous elements in the sequence, but also on future elements in the sequence
  - It resembles two RNNs stacked on top of each other



$$\vec{h}_t = \sigma(\vec{W}^{(hh)}\vec{h}_{t-1} + \vec{W}^{(hx)}x_t)$$

$$\overleftarrow{h}_t = \sigma(\overleftarrow{W}^{(hh)}\overleftarrow{h}_{t+1} + \overleftarrow{W}^{(hx)}x_t)$$

$$y_t = f\left(\left[\vec{h}_t; \overleftarrow{h}_t\right]\right)$$

Outputs both past and future elements

Slide credit: Param Vir Singh – Deep Learning

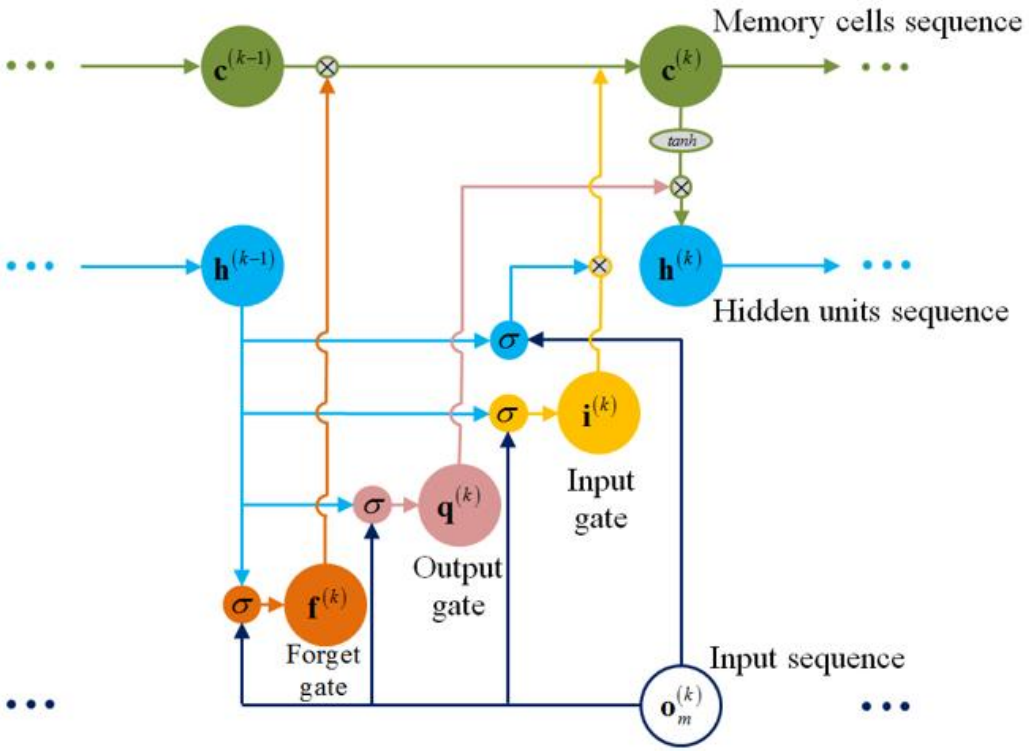
LSTM Networks

- ***Long Short-Term Memory (LSTM)*** networks are a variant of RNNs
- LSTM mitigates the vanishing/exploding gradient problem
  - Solution: a Memory Cell, updated at each step in the sequence
- Three gates control the flow of information to and from the Memory Cell
  - Input Gate: protects the current step from irrelevant inputs
  - Output Gate: prevents current step from passing irrelevant information to later steps
  - Forget Gate: limits information passed from one cell to the next
- Most modern RNN models use either LSTM units or other more advanced types of recurrent units (e.g., GRU units)

LSTM Networks

Recurrent Neural Networks

- LSTM cell
  - Input gate, output gate, forget gate, memory cell
  - LSTM can learn long-term correlations within data sequences



$$\mathbf{i}^{(k)} = \sigma\left(\mathbf{W}_{oi}\mathbf{o}_m^{(k)} + \mathbf{W}_{hi}\mathbf{h}^{(k-1)} + \mathbf{b}_i\right)$$

$$\mathbf{f}^{(k)} = \sigma\left(\mathbf{W}_{of}\mathbf{o}_m^{(k)} + \mathbf{W}_{hf}\mathbf{h}^{(k-1)} + \mathbf{b}_f\right)$$

$$\mathbf{q}^{(k)} = \sigma\left(\mathbf{W}_{oq}\mathbf{o}_m^{(k)} + \mathbf{W}_{hq}\mathbf{h}^{(k-1)} + \mathbf{b}_q\right)$$

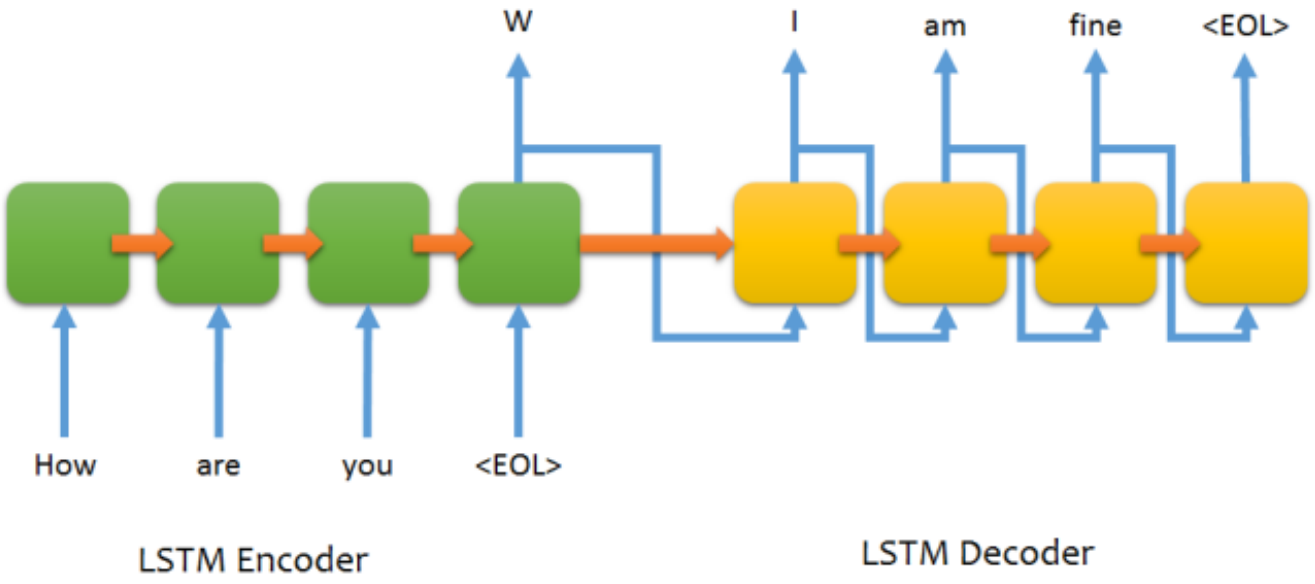
$$\mathbf{c}^{(k)} = \mathbf{f}^{(k)}\mathbf{c}^{(k-1)} + \mathbf{i}^{(k)}\,\sigma\left(\mathbf{W}_{oc}\mathbf{o}_m^{(k)} + \mathbf{W}_{hc}\mathbf{h}^{(k-1)} + \mathbf{b}_c\right)$$

$$\mathbf{h}^{(k)} = \mathbf{q}^{(k)} tanh\left(\mathbf{c}^{(k)}\right)$$

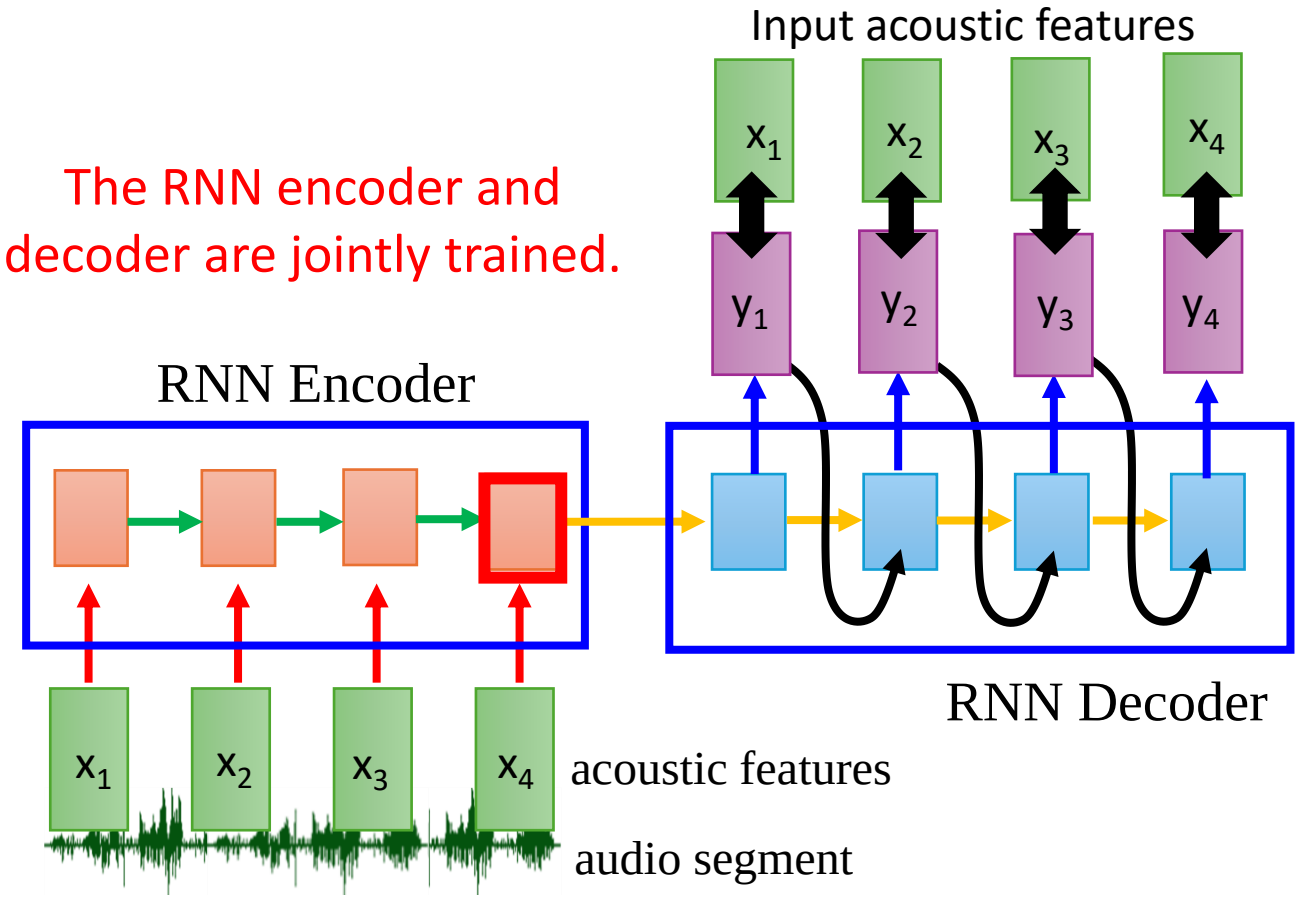
The Encoder-Decoder Model with RNNs

- We introduce a new model, the encoder-decoder model, which is used when we are taking an input sequence and translating it to an output sequence x that is of a different length than the input, and doesn't align with it in a word-to-word way.
- Encoder-decoder models are used especially for tasks like machine translation, where the input sequence and output sequence can have different lengths and the mapping between a token in the input and a token in the output can be very indirect (in some languages the verb appears at the beginning of the sentence; in other languages at the end.)
- Encoder-decoder networks, sometimes called sequence-to-sequence networks, are models capable of generating contextually appropriate, arbitrary length, output sequences given an input sequence. Encoder-decoder networks have been applied to a very wide range of applications including summarization, question answering, and dialogue, but they are particularly popular for machine translation.

Demo: Chat-bot



Sequence-to-sequence Auto-encoder



The encoder-decoder architecture

- The key idea underlying these networks is the use of an encoder network that takes an input sequence and creates a contextualized representation of it, often called the context. This representation is then passed to a decoder which generates a task specific output sequence. Fig. 9.16 illustrates the architecture.

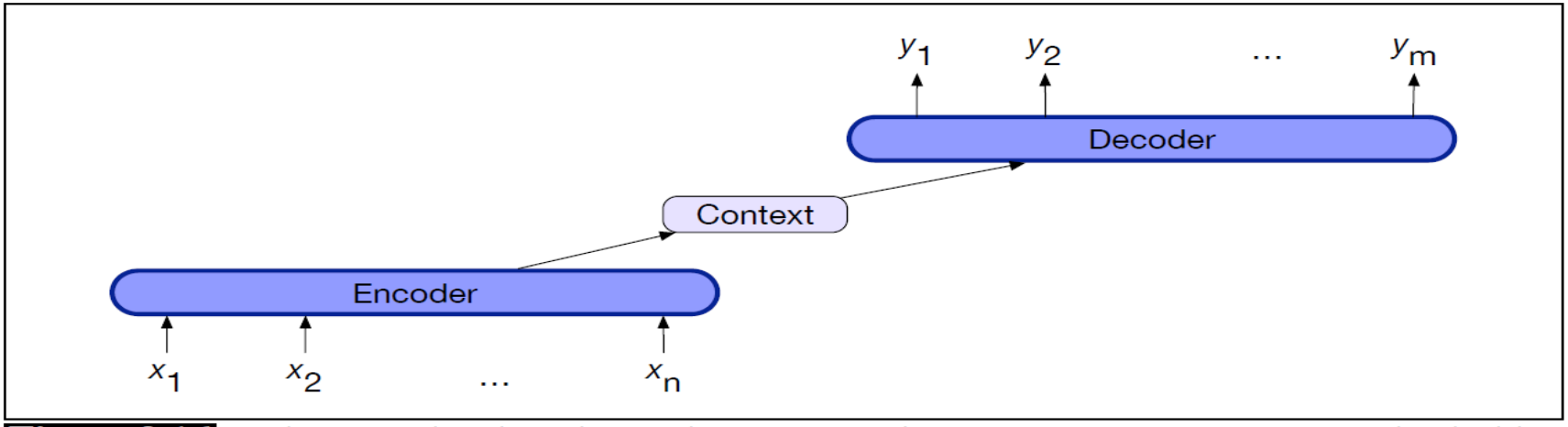


Figure 9.16 The encoder-decoder architecture. The context is a function of the hidden representations of the input, and may be used by the decoder in a variety of ways.

A translation model using an encoder-decoder

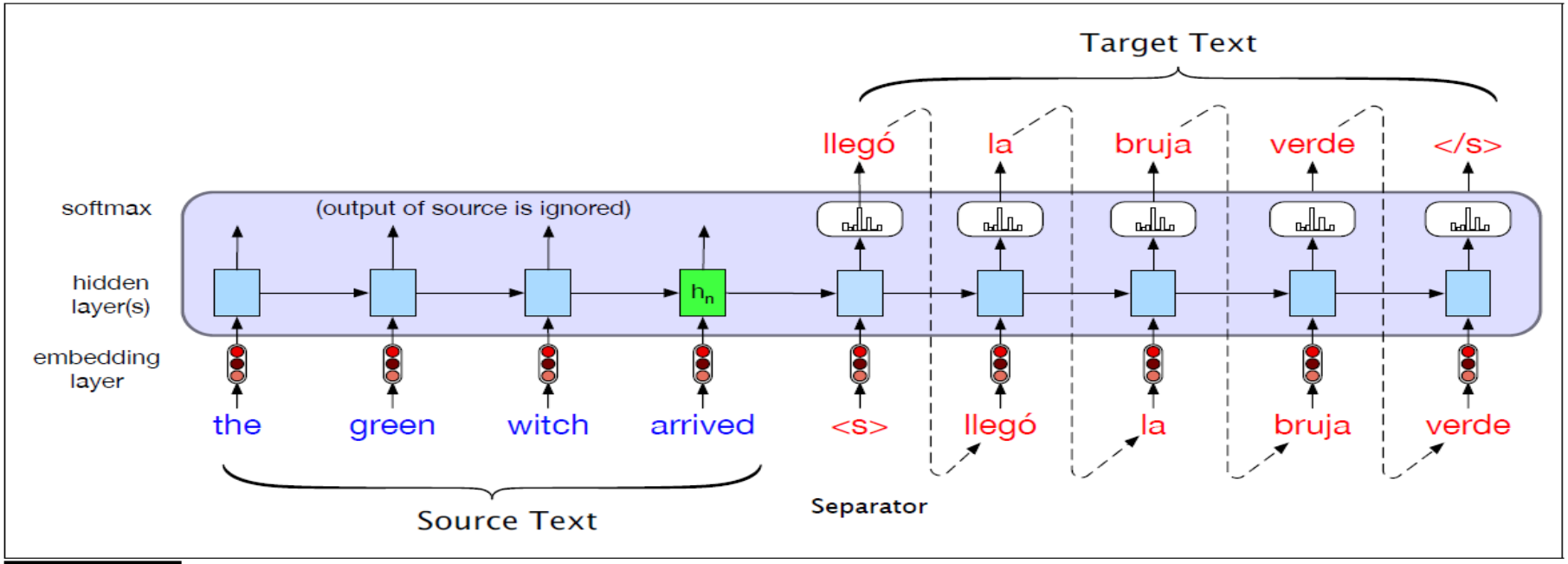


Figure 9.17 Translating a single sentence (inference time) in the basic RNN version of encoder-decoder approach to machine translation. Source and target sentences are concatenated with a separator token in between, and the decoder uses context information from the encoder's last hidden state.

A translation model using an encoder-decoder

- To translate a source text, we run it through the network performing forward inference to generate hidden states until we get to the end of the source.
- Then we begin autoregressive generation, asking for a word in the context of the hidden layer from the end of the source input as well as the end-of-sentence marker.
- Subsequent words are conditioned on the previous hidden state and the embedding for the last word generated.

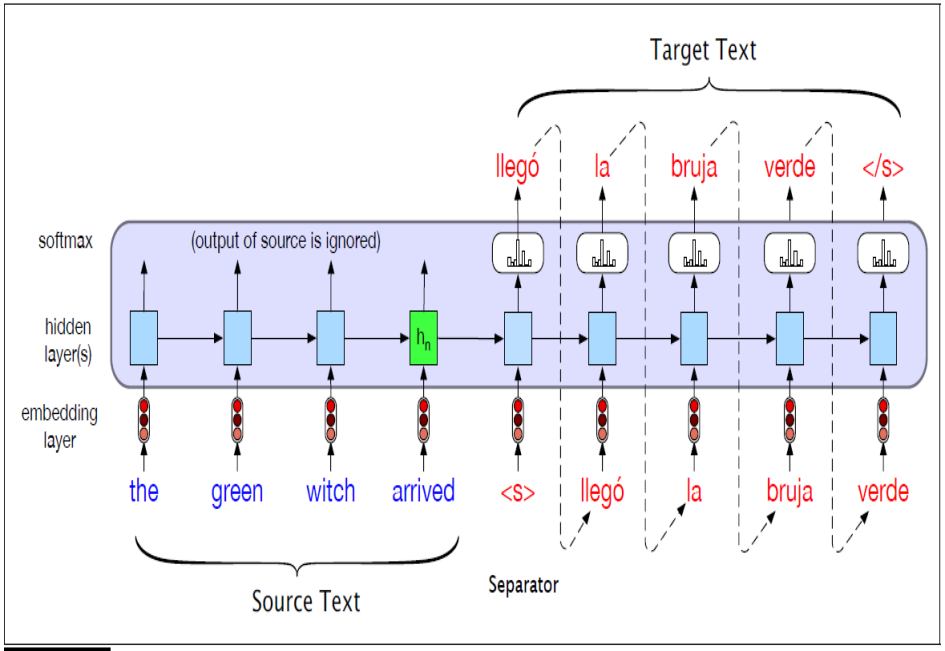
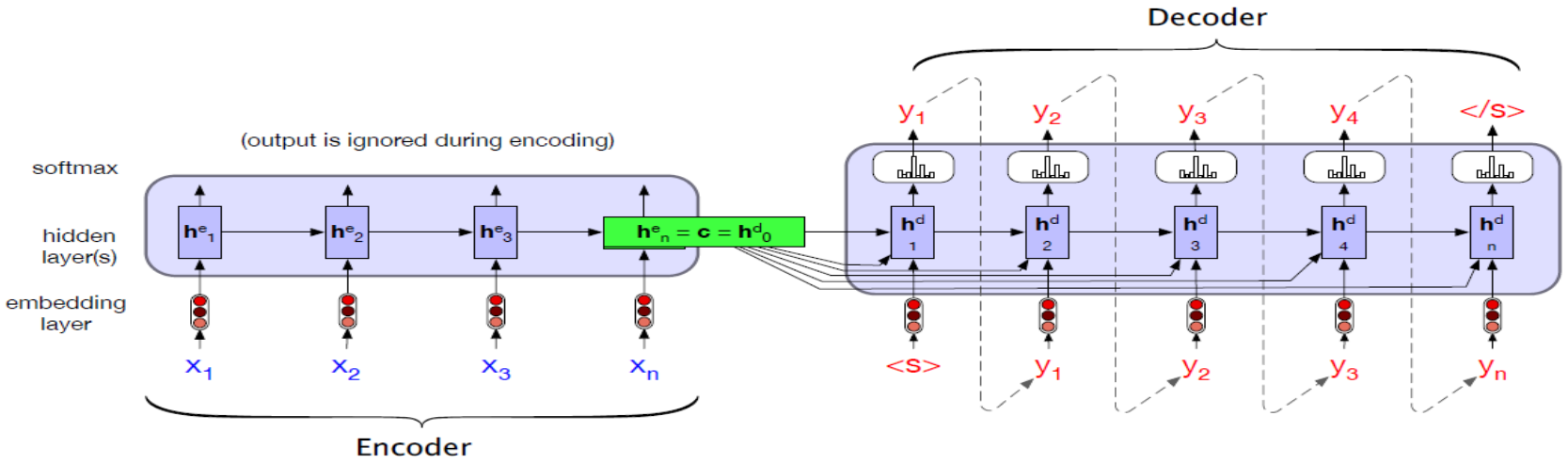


Figure 9.17 Translating a single sentence (inference time) in the basic RNN version of encoder-decoder approach to machine translation. Source and target sentences are concatenated with a separator token in between, and the decoder uses context information from the encoder's last hidden state.

A translation model using an encoder-decoder

- The entire purpose of the encoder is to generate a contextualized representation of the input. This representation is embodied in the final hidden state of the encoder, h_n^e. This representation, also called c for context, is then passed to the decoder.
- The decoder network on the right takes this state and uses it to initialize the first hidden state of the decoder. That is, the first decoder RNN cell uses c as its prior hidden state h_0^d.
- The decoder autoregressively generates a sequence of outputs, an element at a time, until an end-of-sequence marker is generated. Each hidden state is conditioned on the previous hidden state and the output generated in the previous state.



Training the Encoder-Decoder Model

- Encoder-decoder architectures are trained end-to-end, just as with the RNN language models of Chapter 9.
- Each training example is a tuple of paired strings, a source and a target. Concatenated with a separator token, these source-target pairs can now serve as training data.
- For MT, the training data typically consists of sets of sentences and their translations. These can be drawn from standard datasets of aligned sentence pairs.

Training the Encoder-Decoder Model

- The network is given the source text and then starting with the separator token is trained autoregressively to predict the next word, as shown in Fig. 9.20.
- The decoder during inference uses its own estimated output. Thus the decoder will tend to deviate more and more from the gold target sentence as it keeps generating teacher forcing more tokens.
- In training, therefore, it is more common to use teacher forcing in the decoder.
- Teacher forcing means that we force the system to use the gold target token from training as the next input $\mathbf{X}_{t+1}$, rather than allowing it to rely on the (possibly erroneous) decoder output $\hat{y}_t$. This speeds up training.

Training the Encoder-Decoder Model

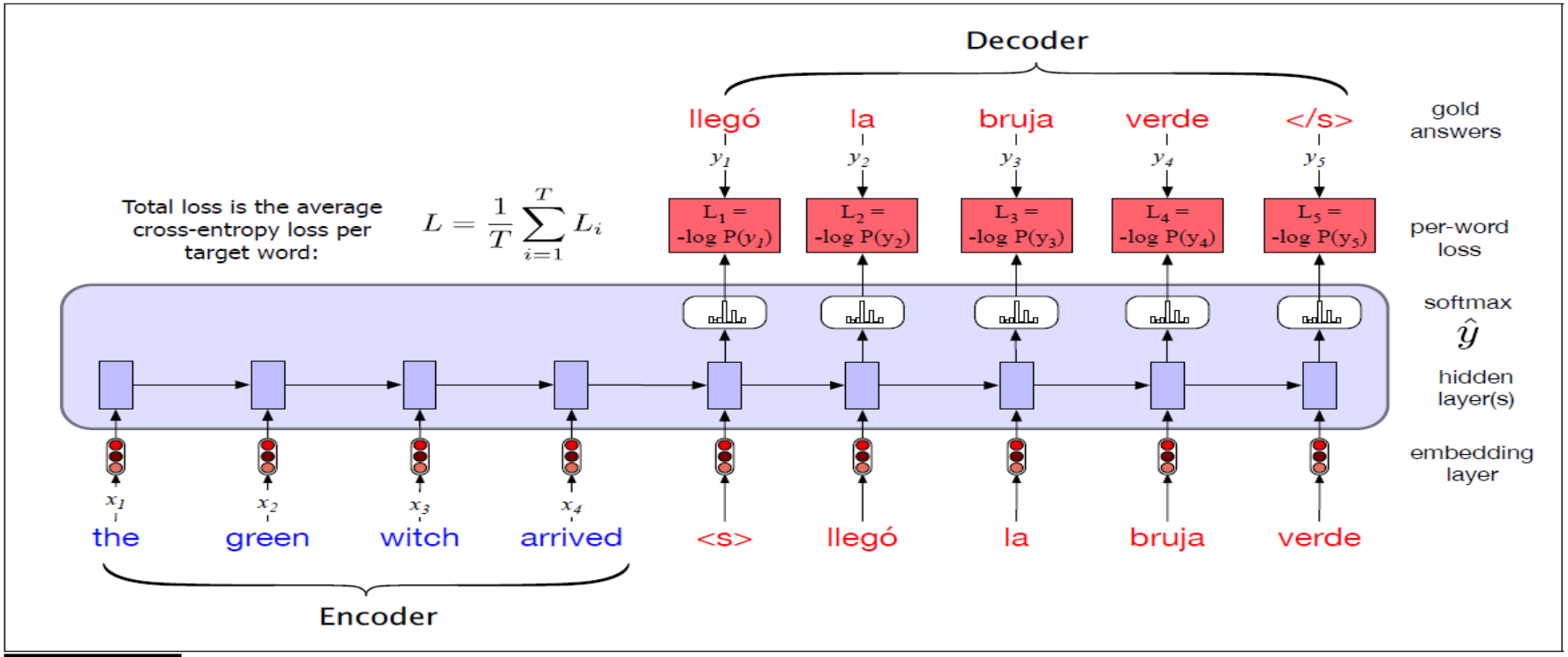


Figure 9.20 Training the basic RNN encoder-decoder approach to machine translation. Note that in the decoder we usually don't propagate the model's softmax outputs $\hat{y}_t$, but use **teacher forcing** to force each input to the correct gold value for training. We compute the softmax output distribution over $\hat{y}$ in the decoder in order to compute the loss at each token, which can then be averaged to compute a loss for the sentence.

Attention

- For many applications, it helps to add “attention” to RNNs.
- Allows network to learn to attend to different parts of the input at different time steps, shifting its attention to focus on different aspects during its processing.
- Used in image captioning to focus on different parts of an image when generating different parts of the output sentence.
- In MT, allows focusing attention on different parts of the source sentence when generating different parts of the translation.

Attention for Image Captioning

(Xu, et al. 2015)

Figure 2. Attention over time. As the model generates each word, its attention changes to reflect the relevant parts of the image. "soft" (top row) vs "hard" (bottom row) attention. (Note that both models generated the same captions in this example.)

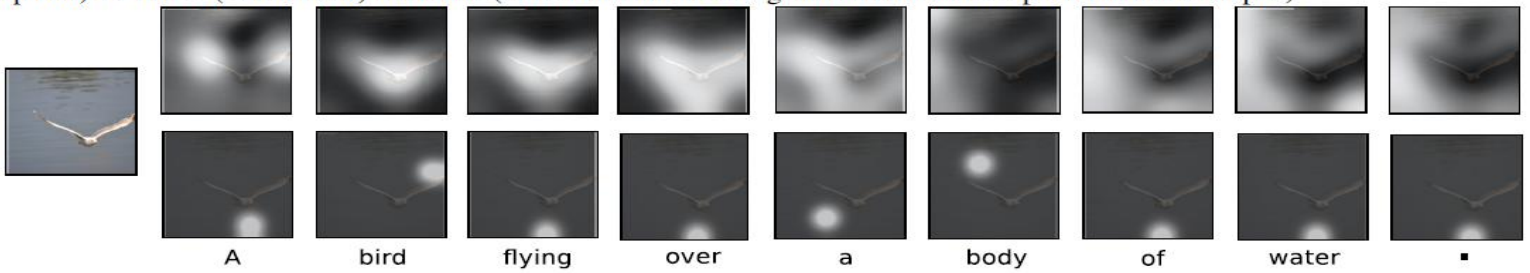


Figure 3. Examples of attending to the correct object (*white* indicates the attended regions, *underlines* indicated the corresponding word)

Attention

- The simplicity of the encoder-decoder model is its clean separation of the encoder— which builds a representation of the source text—from the decoder, which uses this context to generate a target text.
- In the model as we've described it so far, this context vector is h_n, the hidden state of the last (nth) time step of the source text.
- This final hidden state is thus acting as a bottleneck: it must represent absolutely everything about the meaning of the source text, since the only thing the decoder knows about the source text is what's in this context vector (Fig. 9.21).
- Information at the beginning of the sentence, especially for long sentences, may not be equally well represented in the context vector.

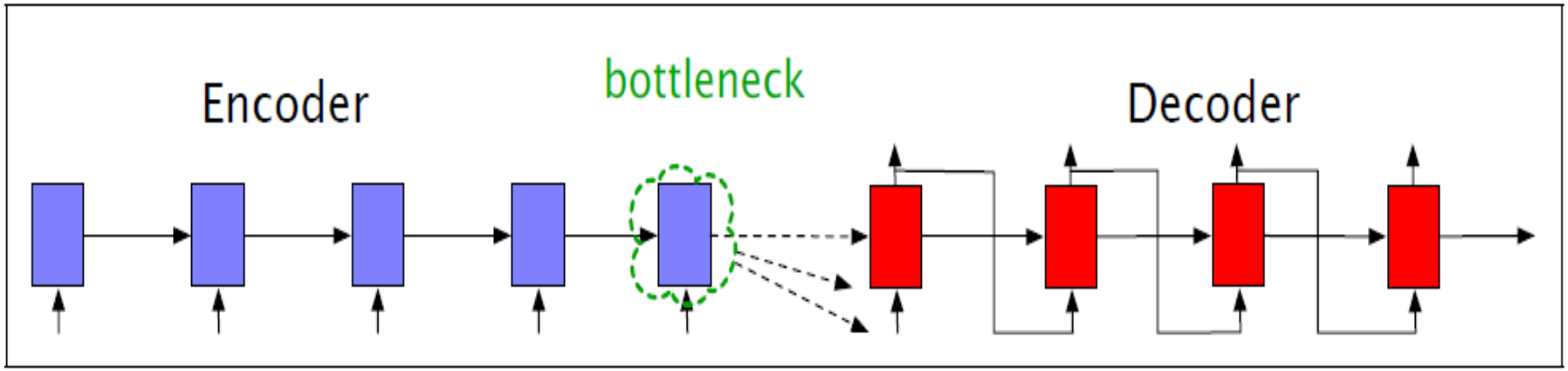


Figure 9.21 Requiring the context *c* to be only the encoder's final hidden state forces all the information from the entire source sentence to pass through this representational bottleneck.

Attention definition

- A mechanism for helping compute the embedding for a token by selectively attending to and integrating information from surrounding tokens (at the previous layer).

- More formally: a method for doing a weighted sum of vectors.

Attention Mechanism

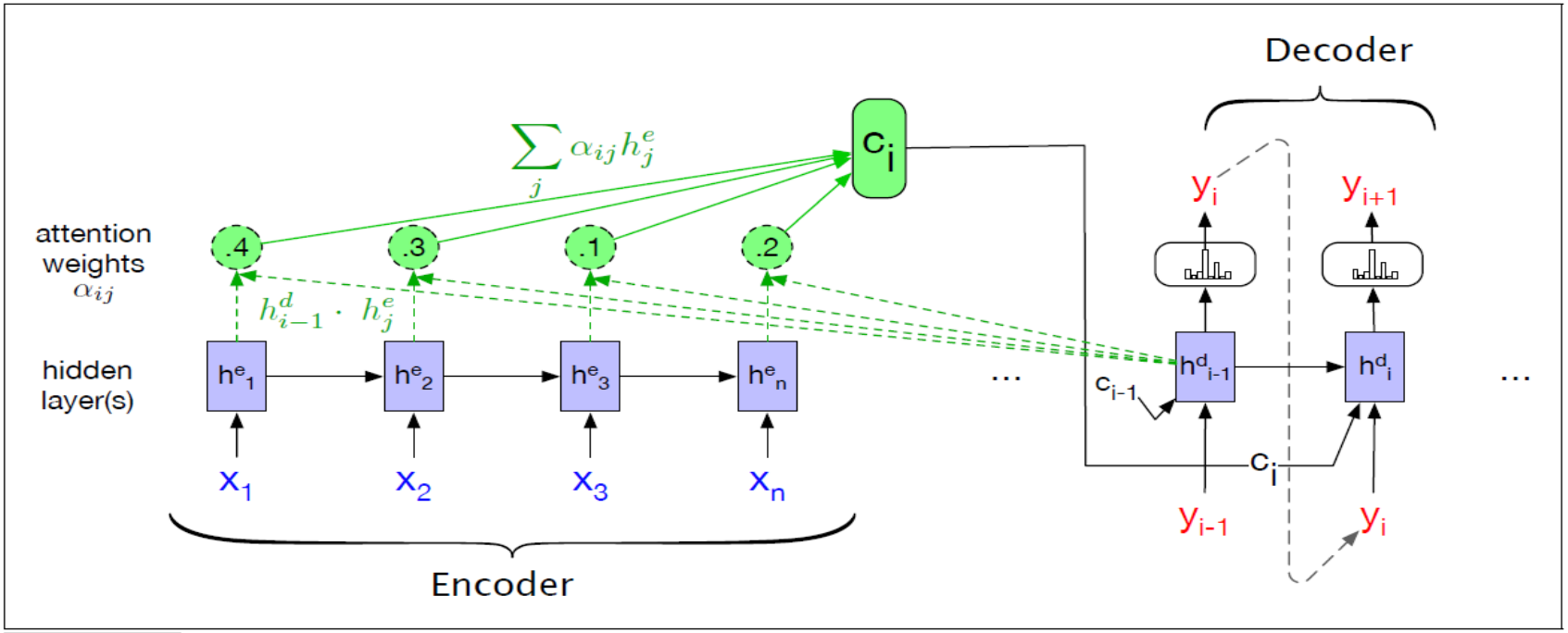
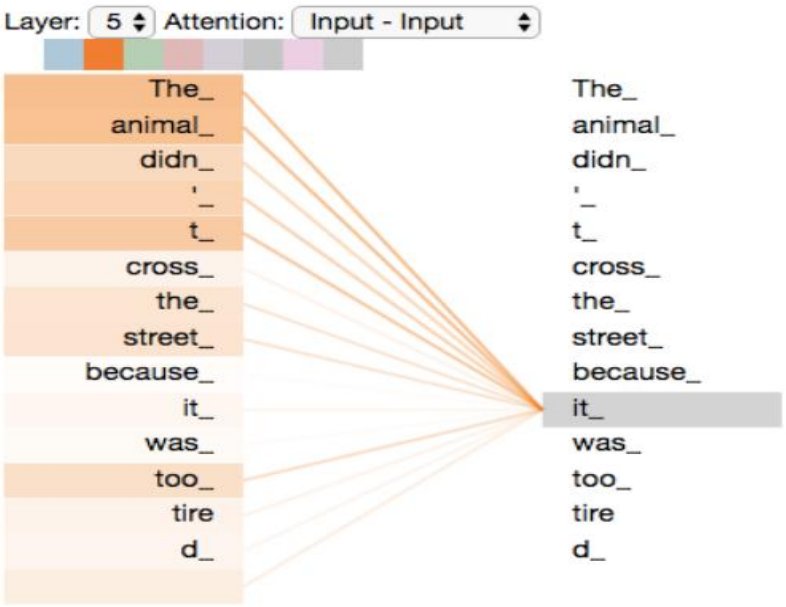


Figure 9.23 A sketch of the encoder-decoder network with attention, focusing on the computation of $\mathbf{c}_i$. The context value $\mathbf{c}_i$ is one of the inputs to the computation of $\mathbf{h}_i^d$. It is computed by taking the weighted sum of all the encoder hidden states, each weighted by their dot product with the prior decoder hidden state $\mathbf{h}_{i-1}^d$.

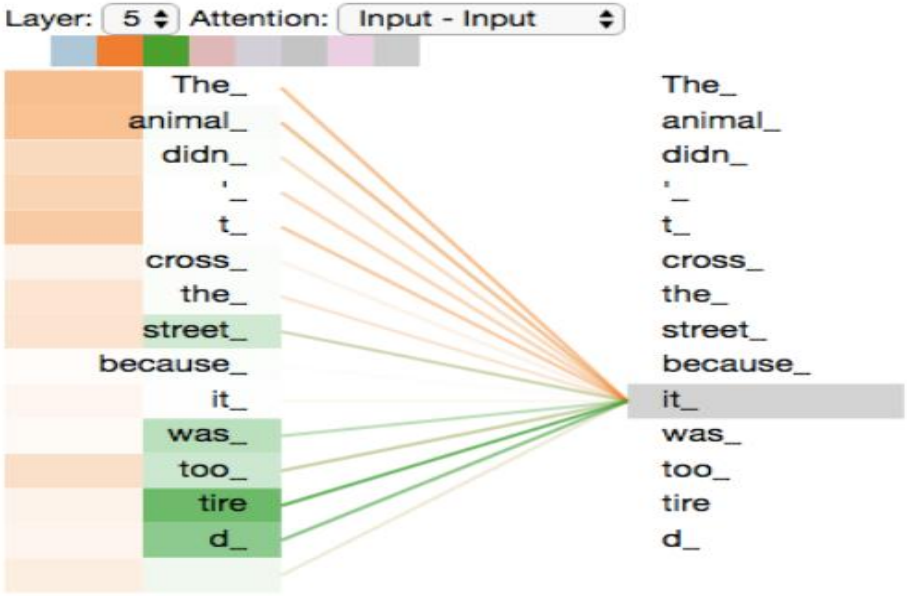
Multihead Attention

- The different words in a sentence can relate to each other in many different ways simultaneously. For example, distinct syntactic, semantic, and discourse relationships can hold between verbs and their arguments in a sentence.
- It would be difficult for a single transformer block to learn to capture all of the different kinds of parallel relations among its inputs.
- Transformers address this issue with multihead self-attention layers.
- These are sets of self-attention layers, called heads, that reside in parallel layers at the same depth in a model, each with its own set of parameters.
- Given these distinct sets of parameters, each head can learn different aspects of the relationships that exist among inputs at the same level of abstraction.

Multihead Attention



HEAD 1: As we are encoding the word "it" in encoder #5 (the top encoder in the stack), part of the attention mechanism was focusing on "The Animal", and baked a part of its representation into the encoding of "it".

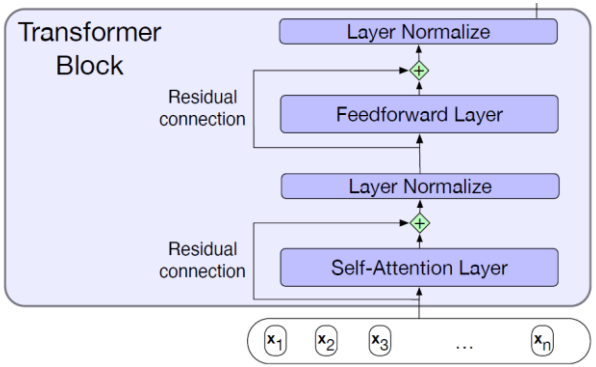


HEAD 2: As we encode the word "it", one attention head is focusing most on "the animal", while another is focusing on "tired" -- in a sense, the model's representation of the word "it" bakes in some of the representation of both "animal" and "tired".

Source: https://jalammar.github.io/illustrated-transformer/

Transformers

- Transformers map sequences of input vectors (x_1....,x_n) to sequences of output vectors (y_1,...,y_n) of the same length.
- Transformers are made up of stacks of transformer blocks, each of which is a multilayer network made by combining simple linear layers, feedforward networks, and **self-attention layers**, the key innovation of transformers.
- Self-attention allows a network to directly extract and use information from arbitrarily large contexts without the need to pass it through intermediate recurrent connections as in RNNs.



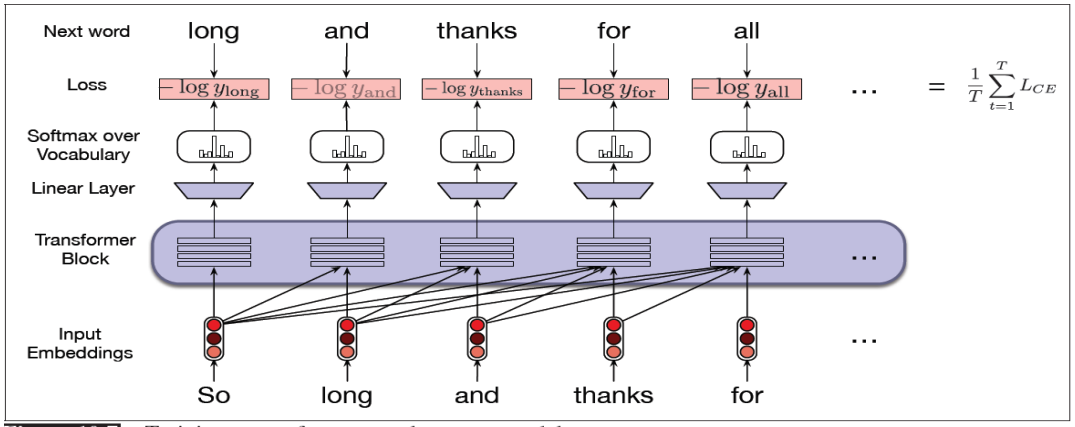


Figure 10.7 Training a transformer as a language model.

Summary

- Transformers are non-recurrent networks based on **self-attention**. A self-attention layer maps input sequences to output sequences of the same length, using attention heads that model how the surrounding words are relevant for the processing of the current word.
- A transformer block consists of a single attention layer followed by a feed-forward layer with residual connections and layer normalizations following each. Transformer blocks can be stacked to make deeper and more powerful networks.
- Common language-based applications for RNNs and transformers include:
  - Probabilistic language modeling: assigning a probability to a sequence, or to the next element of a sequence given the preceding words.
  - Auto-regressive generation using a trained language model.
  - Sequence labeling like part-of-speech tagging, where each element of a sequence is assigned a label.
  - Sequence classification, where an entire text is assigned to a category, as in spam detection, sentiment analysis or topic classification.

Transformers as Language Models

- We've seen all the major components of transformers, let's examine how to deploy them as language models via self-supervised learning.
- Given a training corpus of plain text we'll in a sequence y_t , using cross-entropy loss. train the model autoregressively to predict the next token (Recall from Section 9.3.3 that using a language model to incrementally generate words by repeatedly sampling the next word conditioned on our previous choices is called **autoregressive generation** or **causal LM generation**.)
- Fig. 10.7 illustrates the general training approach. At each step, given all the preceding words, the final transformer layer produces an output distribution over the entire vocabulary. During training, the probability assigned to the correct word is used to calculate the cross-entropy loss for each item in the sequence

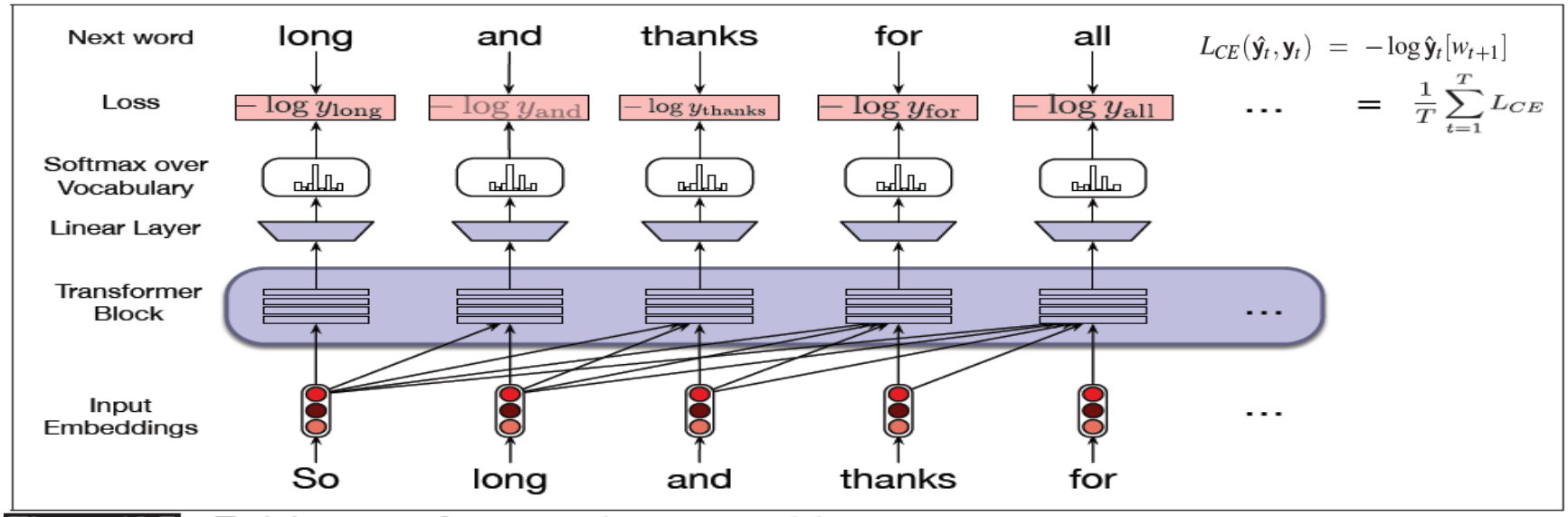
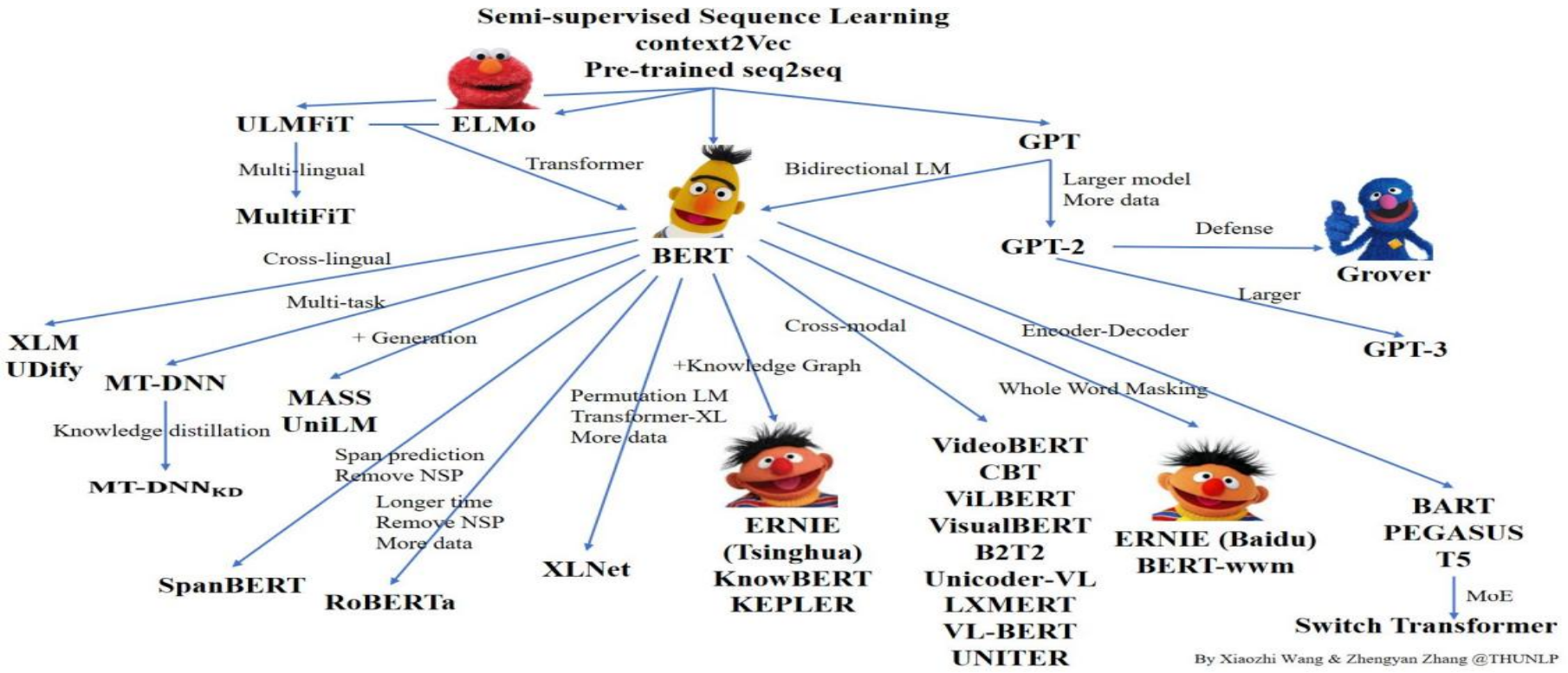


Figure 10.7 Training a transformer as a language model.

Explosion of Pre-trained LMs



References

1. Hung-yi Lee – Deep Learning Tutorial
2. Ismini Lourentzou – Introduction to Deep Learning
3. CS231n Convolutional Neural Networks for Visual Recognition (Stanford CS course) (link)
4. James Hays, Brown – Machine Learning Overview
5. Param Vir Singh, Shunyuan Zhang, Nikhil Malik – Deep Learning
6. Sebastian Ruder – An Overview of Gradient Descent Optimization Algorithms (link)