ASTR8150/PHYS8150
Neural Networks

Fabien Baron

Georgia State University

fbaron@gsu.edu

Fall 2025

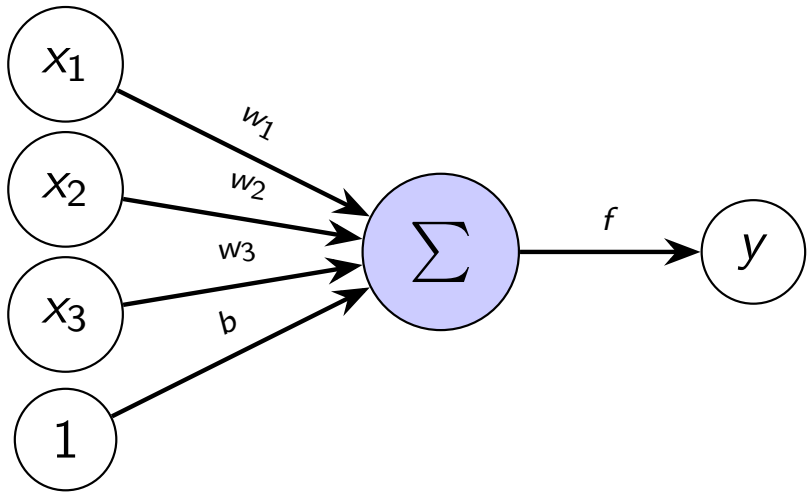
What is a Neural Network?

- A neural network is a computational model inspired by the way biological neural networks in the brain process information.
- Neural network "models" consist of layers of interconnected nodes (neurons) that can learn complex patterns from data
- A well-designed network can approximate any continuous function on compact (finite) domains. Better approximations will require larger networks.
- Models are optimized by adjusting weights based on errors between predicted and actual values, using gradient descent variants adapted to large number of parameters.
- These models are the foundation of deep learning models
- Typical tasks: classification, regression, pattern recognition, regularization, generation of patterns.

What is a neuron?

- A neuron, is a weighted sum of the inputs, followed by an activation function.
- A neuron first computes a linear transformation known as preactivation:

$$a = \sum_i w_i x_i + b$$

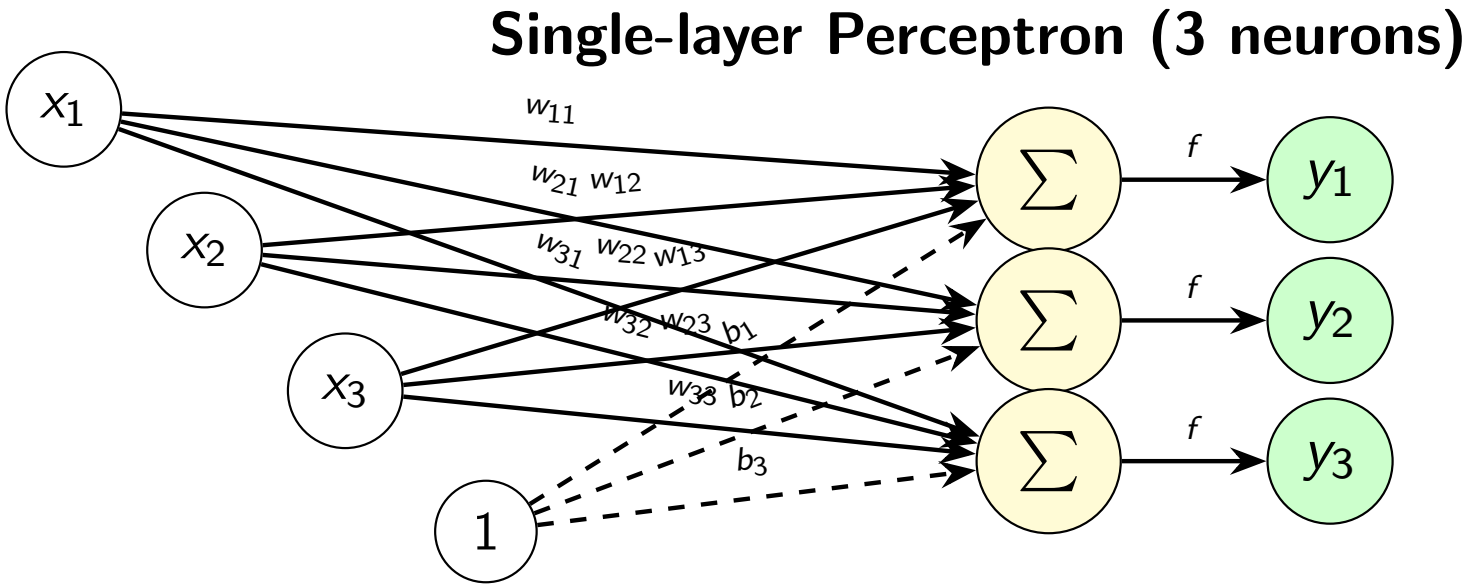


- Nonlinearity is introduced by using an activation function f:

$$y = f(a)$$

- The single-perceptron was the first neuron (developed for classification by Rosenblatt in 1958).

The Single-layer Perceptron



- Single layer of neurons that compute a weighted sum of inputs and apply an activation
- Suitable for classification tasks.

Weights and Biases

- **Weights:** Parameters that determine the strength of the connection between neurons.
- **Biases:** the bias $b = w_0$ allows shifting of activation thresholds and this added flexibility helps the model make better predictions.
- The output of a neuron is calculated as:

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b\right)$$

where:

- w_i are the weights
- x_i are the inputs
- b is the bias
- f is the activation function

Activation Functions

- Activation functions introduce nonlinearity and enable networks to model complex mappings. Which ones to use depend on what the layer is designed to do (comes with experience).
- **Sigmoid:**

$$f(x) = \frac{1}{1 + e^{-x}}$$

  Maps input values to a range between 0 and 1.
- **Tanh:**

$$f(x) = \tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

  Maps input values to a range between -1 and 1.
- **ReLU (Rectified Linear Unit):**

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

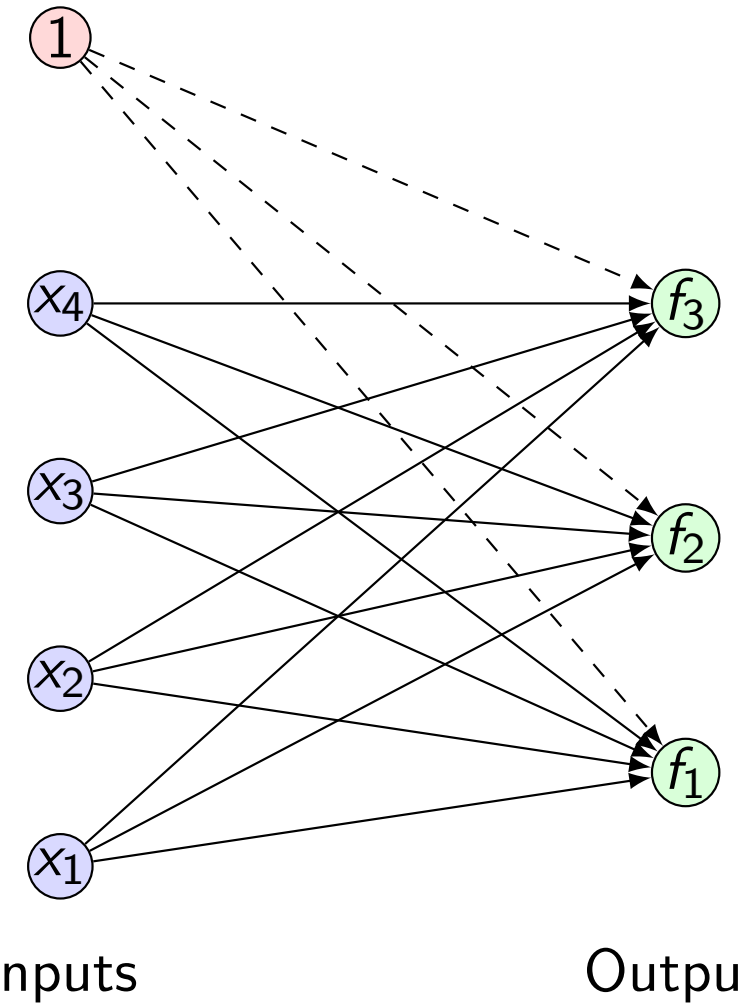
  Allows only positive values to pass through.
- **Softmax:**

$$f(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

Dense Layers

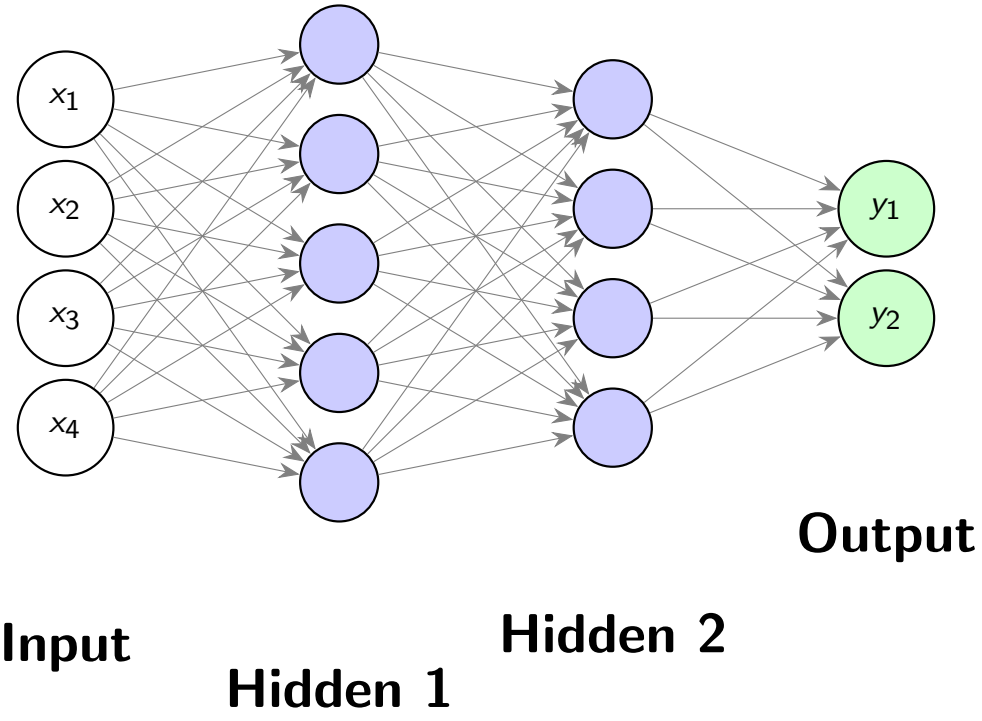
- A **dense layer** (fully connected layer) connects every neuron in the current layer to every neuron in the next layer.
- Each neuron in a layer is connected to neurons in adjacent layers through weighted connections.
- Each connection has an associated weight and each neuron has an associated bias.

Dense Layer



Neural network architecture: the Multilayer Perceptron

- A neural network consists of three types of layers
- **Input layer:** Receives the input data (contain features).
- **Hidden layers:** Intermediate layers where features are extracted but outputs are not directly observed.
- **Output layer:** Produces the final output (prediction).



- MLPs can model complex relationships and solve problems that are not linearly separable.
- MLPs are widely used in tasks like image recognition, speech processing, and time series prediction.

Training Neural Networks

- The goal of training a neural network is to minimize the error between the predicted and actual outputs.
- Training involves:
  - Feeding input data into the network.
  - Computing the output through forward propagation.
  - Comparing the output with the true label (using a loss function).
  - Adjusting weights to minimize the error using an optimization algorithm.
- Loss functions depend on the task
  - Regression: squared error.
  - Classification: cross-entropy.
- **Epochs:** The number of times the network is trained on the entire dataset.

Gradient Descent for neural networks

- Variants of gradient descent are used to minimize the loss function.
- The weight update rule for gradient descent is:

$$w_i = w_i - \eta \frac{\partial L}{\partial w_i}$$

where:

  - η is the learning rate.
  - $\frac{\partial L}{\partial w_i}$ is the gradient of the loss function with respect to weight w_i.
- There are different variants of gradient descent:
  - **Stochastic Gradient Descent (SGD):** Uses a single data point for each update.
  - **Batch Gradient Descent:** Uses the entire dataset for each update.
  - **Mini-batch Gradient Descent:** Uses a small batch of data for each update.

Backpropagation

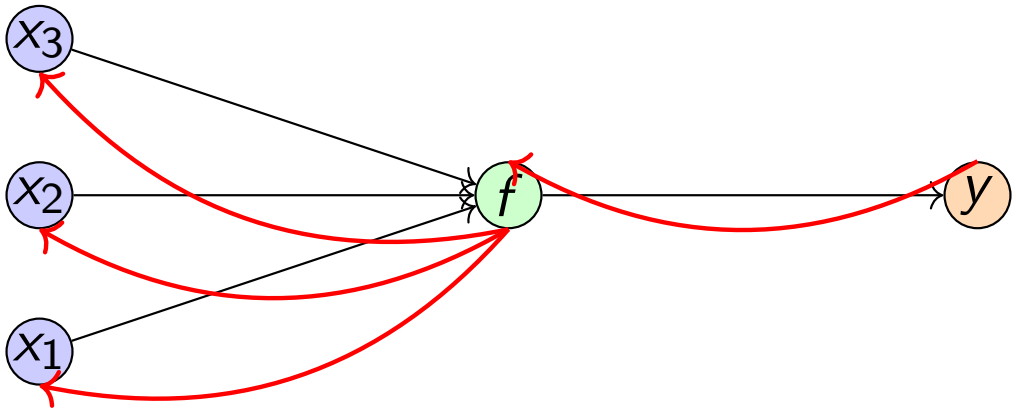
- How do we calculate the gradient of the network with respect to weights? Via backpropagation.
- This led to the development of automatic differentiation
- The algorithm involves two main steps:
  - **Forward Propagation:** Input data is passed through the network to get the output.
  - **Backward Propagation:** The error is calculated, and the gradients of the loss function with respect to the weights are computed.

Backpropagation: Intuition

- Goal: Compute $\frac{\partial L}{\partial w_{ij}}$ for all weights w_{ij} in the network.
- Uses the **chain rule** to propagate gradients from the output layer back to hidden layers.
- For a single hidden neuron f_j:

$$\frac{\partial L}{\partial w_{ji}} = \frac{\partial L}{\partial f_j} \frac{\partial f_j}{\partial a_j} \frac{\partial a_j}{\partial w_{ji}}$$

where $a_j = \sum_i w_{ji} x_i + b_j$ is the pre-activation.



Backpropagation: Algorithm

1. **Forward pass:** Compute all activations a_j and outputs f_j, y_k.
2. **Compute output layer gradient:**

$$\delta_k = \frac{\partial L}{\partial y_k} \odot g'(a_k)$$

   where g is the **output layer activation function** (e.g., softmax for classification).
3. **Propagate to hidden layers:**

$$\delta_j = f'(a_j) \sum_k w_{kj} \delta_k$$

4. **Compute weight gradients:**

$$\frac{\partial L}{\partial w_{ji}} = \delta_j x_i$$

5. **Update weights:**

$$w_{ji} \leftarrow w_{ji} - \eta \frac{\partial L}{\partial w_{ji}}$$

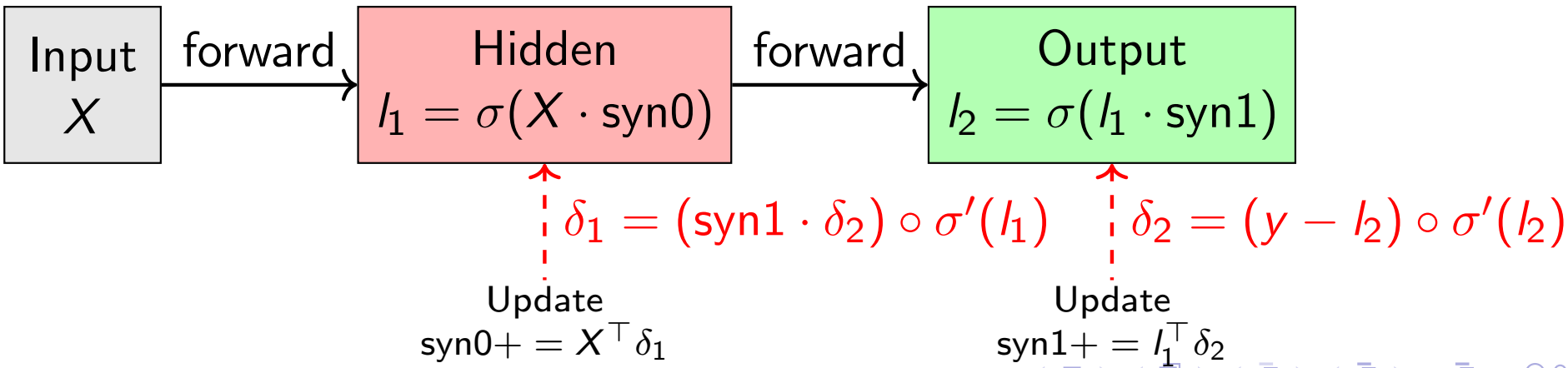
Backpropagation in Trask Toy Network (Sigmoid Activation)

- Input $X \to$ Hidden $l_1 = \sigma(X \cdot \text{syn0}) \to$ Output $l_2 = \sigma(l_1 \cdot \text{syn1})$
- Forward pass computes activations.
- Output error: $\text{error}_2 = y - l_2$
- Backprop computes:

$$\delta_2 = \text{error}_2 \circ \sigma'(l_2), \quad \delta_1 = (\text{syn1} \cdot \delta_2) \circ \sigma'(l_1)$$

- Weight updates:

$$\text{syn1} += l_1^\top \delta_2, \quad \text{syn0} += X^\top \delta_1$$



From MLPs to CNNs

- MLPs (fully connected/dense layers):
  - Flatten image into a long vector; some of the spatial layout is lost.
  - Many parameters and this scales with input size.
- Convolutional Neural Networks (CNNs):
  - Local connectivity: neurons look at small spatial patches.
  - Weight sharing: same convolution filter applied across the image.
  - Preserve H×W layout and are parameter-efficient.

Example: RGB image 32×32×3

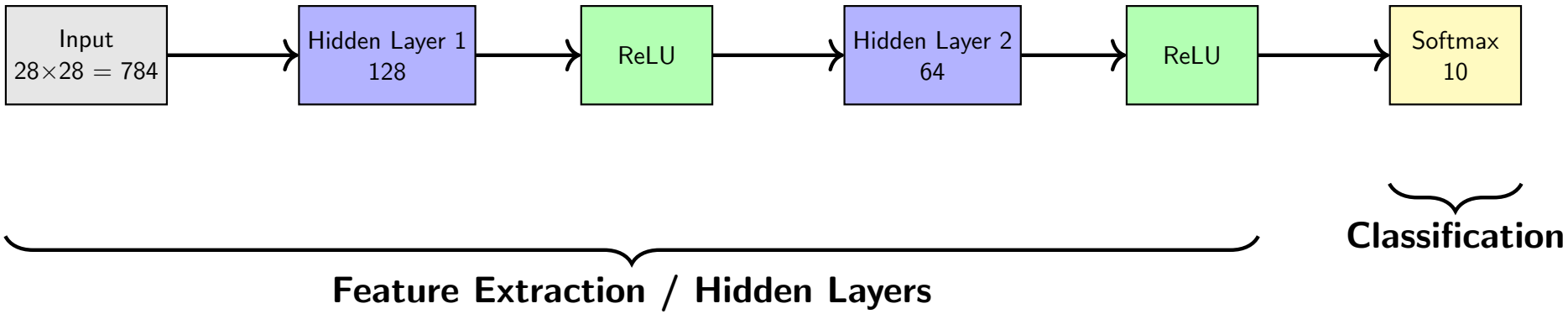
Dense layer (100 units):
parameters $= (32 \cdot 32 \cdot 3) \cdot 100 \approx 307{,}200$

Conv layer (3×3 kernels, 64 filters):
parameters $= 3 \cdot 3 \cdot 3 \cdot 64 = 1{,}728$

$\Rightarrow$ Conv layer uses $\sim$**180× fewer** params

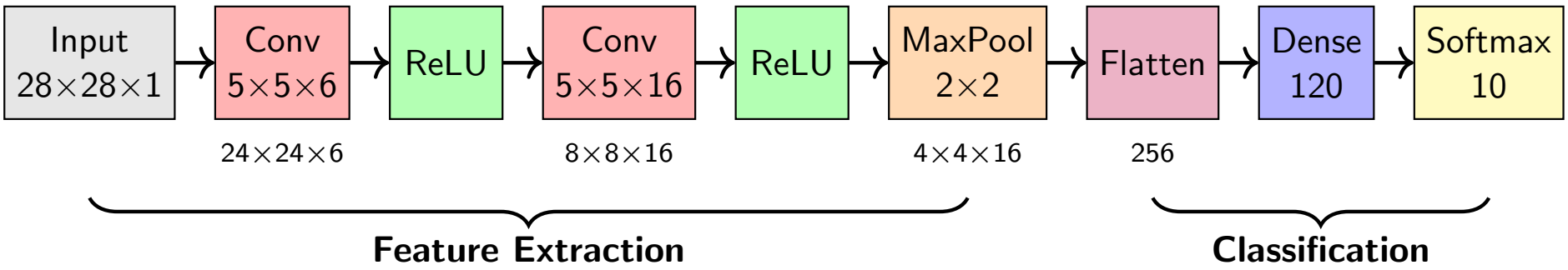
Basic MLP architecture (MNIST classification)

- Fully connected layers followed by activation functions
- Classification done by softmax
- Example below for MNIST classification



Basic CNN architecture (MNIST classification)

- Typical block: [Conv → ReLU → Conv → ReLU → Pool]
- Repeat several blocks to build hierarchy.
- Final: [Flatten → Dense layer → Softmax] for classification
- Design choices: kernel sizes, number of filters, strides, pooling policy.
- Example below for MNIST classification



Convolution and Feature Maps

- Convolution extracts local patterns (edges, textures) from input image patches.
- Just a discrete convolution (no flip, implemented as cross-correlation in most libraries)!
- Kernel slides over the image (stride) producing a smaller output (downsampling).
- Each kernel has depth = number of input channels; sums over channels to create a single feature map.
- Stacking multiple kernels produces multiple feature maps.
- Each kernel produces one feature map; multiple kernels produce multiple maps.

Convolution and Feature Maps

- Input: an image of size $H \times W \times C$, where H = height, W = width, C = number of channels (e.g., 3 for RGB).
- A convolutional kernel/filter of size $k \times k$ has depth = C (matches the input channels).
- At each spatial location, the kernel performs an element-wise multiplication with the corresponding $k \times k \times C$ patch of the input, then sums over all elements to produce a single scalar.
- This scalar becomes one element of the output feature map.
- The kernel slides across the input with a given **stride** S, producing a smaller output in height and width:

$$H_{\text{out}} = \frac{H - k + 2P}{S} + 1, \quad W_{\text{out}} = \frac{W - k + 2P}{S} + 1$$

  where P is the padding applied to the input.
- Each kernel produces one feature map of size $H_{\text{out}} \times W_{\text{out}}$.
- Multiple kernels (K of them) produce K feature maps, which can be stacked to form an output tensor of size $H_{\text{out}} \times W_{\text{out}} \times K$.

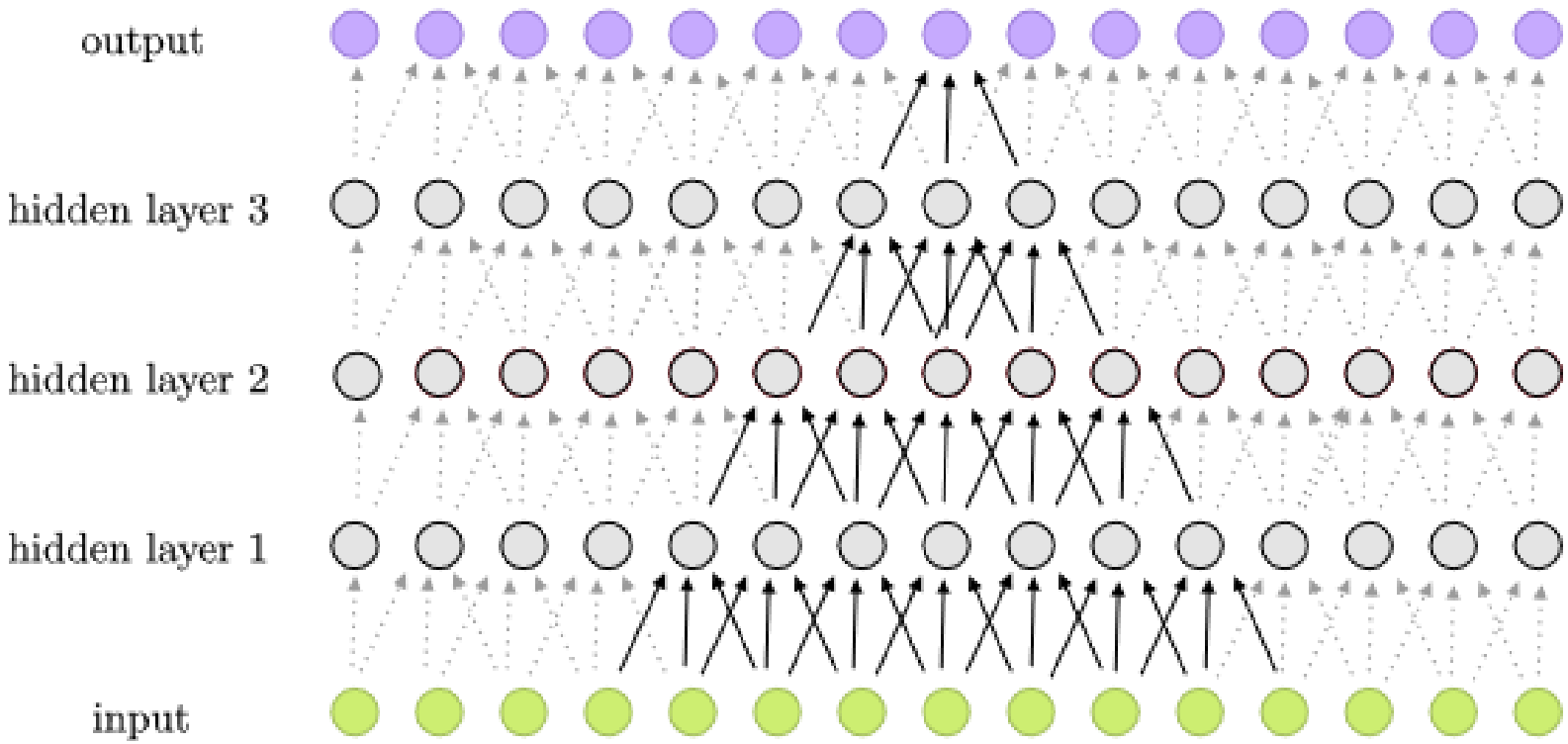
Convolution, then pooling

- The convolution unit is the first building block of a CNN. It requires some kernel specifications:
  - Padding P: add border (often zeros) to control edge behavior.
  - Stride S: how far the kernel moves each step.
  - Output size:

$$H_{\text{out}} = \left\lfloor \frac{H - k + 2P}{S} \right\rfloor + 1.$$

- Pooling reduces spatial dimensions and keeps important signals.
  - Max pooling (e.g. 2×2, stride 2) picks the strongest activation in each window.
  - Average pooling keep the average of the window.
  - Reduces the number of variables: faster training.

1D CNN Example: Feed-forward Illustration



- This diagram shows a 3-layer "1D" convolutional network with kernel size = 3 and stride = 1.
- Convolution in each layer: window slides over the input, producing a feature map. After convolution, the activation function is applied (not shown).

Receptive Field in CNNs

- The receptive field of a neuron is the region of the previous layer it "sees."
- In dense (fully connected) layers, each neuron sees the entire previous layer.
- In a convolutional layer, each neuron sees only a local patch (e.g., 5×5).
- Stacking convolutional layers increases the effective receptive field:
  - Neurons consider larger areas of the input progressively.
- "Dilated convolutions" expand receptive field without increasing parameters:
  - Sparse sampling of input
  - Can combine multiple dilation rates for variable receptive field sizes

Flattening in CNNs

- Flattening is typically used right before a dense classification layer in CNNs.
- Fully connected/dense layers expect a 1D vector input.
- Convolution and pooling layers output the feature maps with shape $H \times W \times C$, $H =$ height, $W =$ width, $C =$ number of channels
- Flattening therefore reshapes the 3D feature maps into a 1D vector:

$$\text{Flatten} : H \times W \times C \quad \rightarrow \quad H \cdot W \cdot C$$