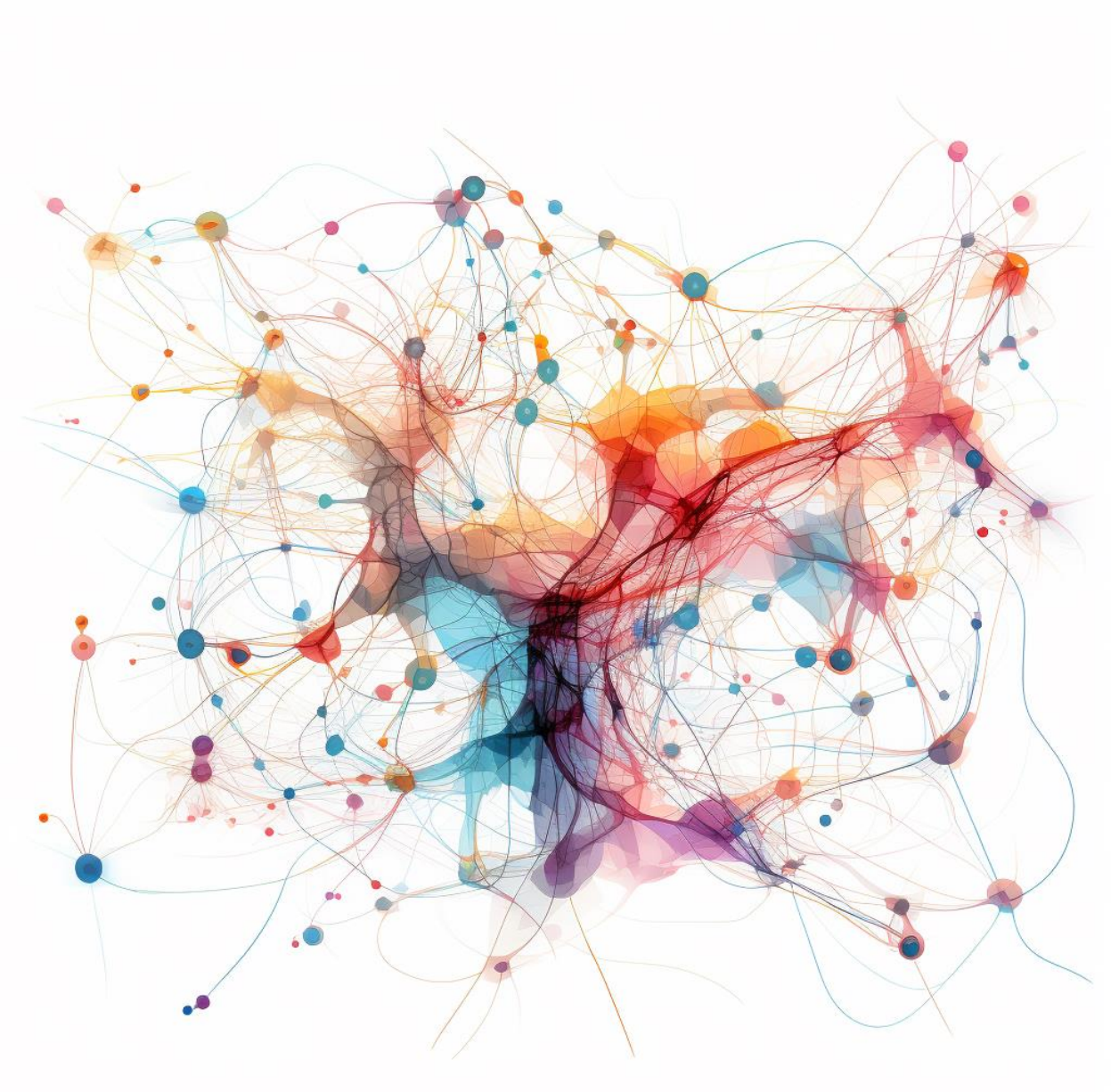
DEEP LEARNING



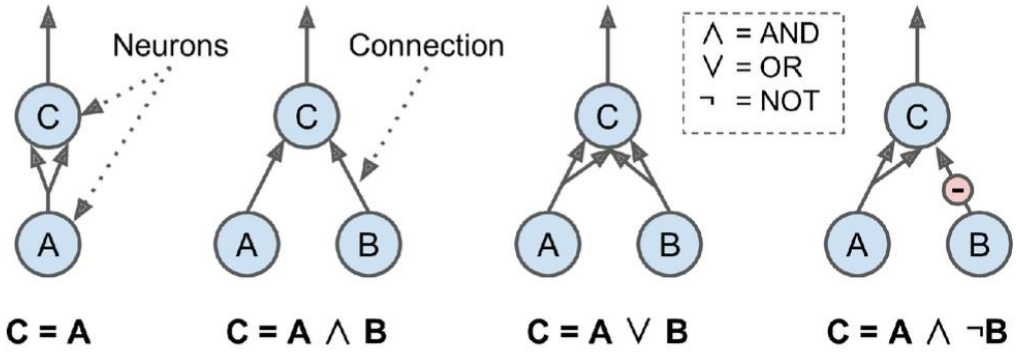
Intro

ANNs are at the very core of Deep Learning. They are versatile, powerful, and scalable, making them ideal to tackle large and highly complex Machine Learning tasks

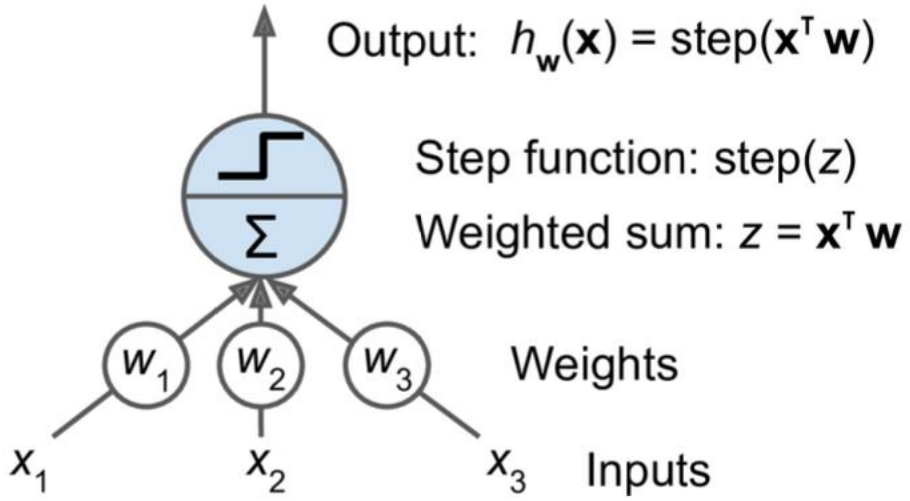
The current resurgence of interest in Artificial Neural Networks (ANNs) is believed to be different and more impactful than previous waves for several reasons:

1. The availability of massive datasets allows ANNs to excel in solving complex problems, often outperforming other machine learning techniques.
2. Increase in computing power, attributed to advancements in technology (Moore's law) and the development of powerful GPU cards driven by the gaming industry, enables the training of large neural networks in reasonable timeframes. Cloud platforms further democratize access to this computing power.
3. While training algorithms have only been slightly modified since the 1990s, these adjustments have significantly enhanced their effectiveness.
4. Concerns about ANNs getting stuck in local optima have proven to be less problematic in practice than anticipated.

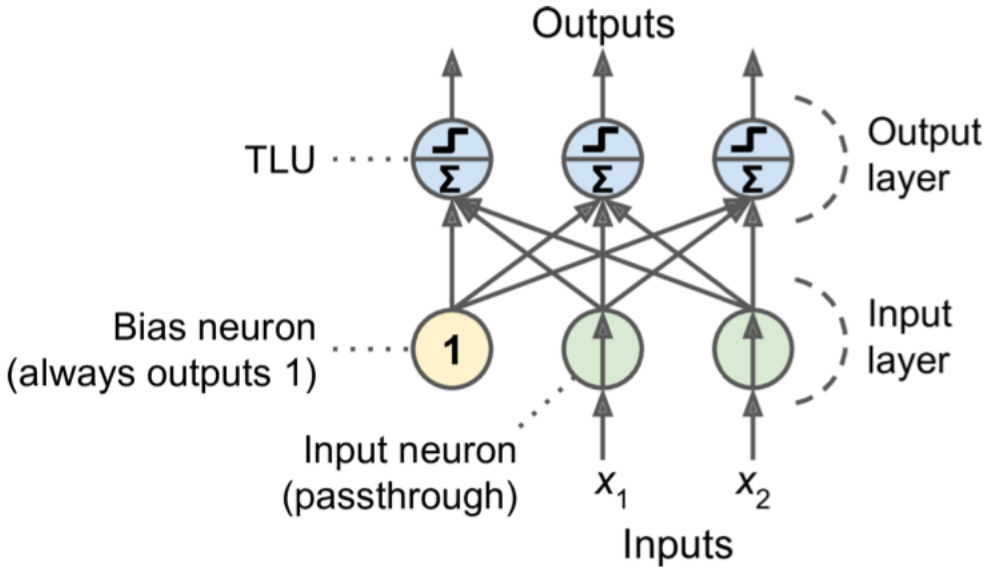
Logical Computations with Neurons



The Perceptron



The Perceptron, invented in 1957 by Frank Rosenblatt, is a foundational artificial neural network (ANN) architecture. It utilizes a threshold logic unit (TLU) or linear threshold unit (LTU) to process inputs, which are numbers associated with weights. The TLU calculates a weighted sum of these inputs and applies a step function, typically the Heaviside or sign function, to produce an output. This mechanism allows for simple linear binary classification by evaluating if the computed sum exceeds a certain threshold, designating the input as belonging to one of two possible classes.

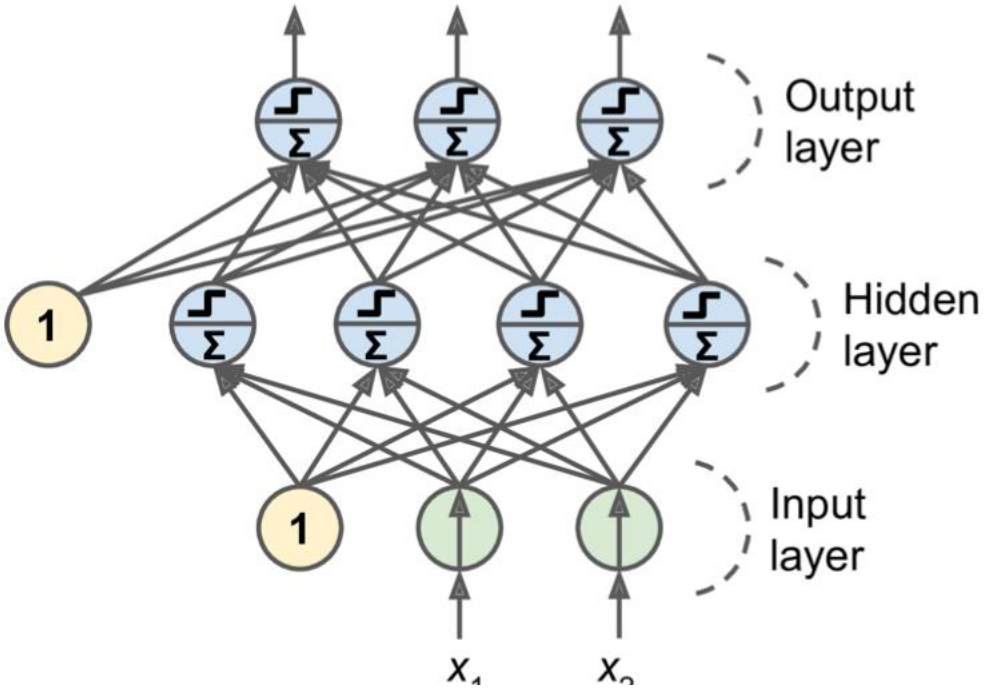


A Perceptron consists of a single layer of TLUs, each connected to all inputs, forming a fully connected or dense layer. Inputs pass through special neurons (input neurons and a bias neuron) to ensure consistent processing. This structure enables the Perceptron to act as a multioutput classifier capable of distinguishing between multiple binary classes simultaneously.

Training a Perceptron follows a rule inspired by Hebb's theory, suggesting that neurons that fire together strengthen their connection. This learning rule adjusts the weights based on the prediction error for each training instance, reinforcing weights that would have contributed to a correct prediction. Despite its simplicity and linear decision boundary, the Perceptron can converge to a solution for linearly separable instances, a principle known as the Perceptron convergence theorem.

You may have noticed that the Perceptron learning algorithm strongly resembles Stochastic Gradient Descent. Note that contrary to Logistic Regression classifiers, Perceptrons do not output a class probability; rather, they make predictions based on a hard threshold. This is one reason to prefer Logistic Regression over Perceptrons.

The Multilayer Perceptron and Backpropagation

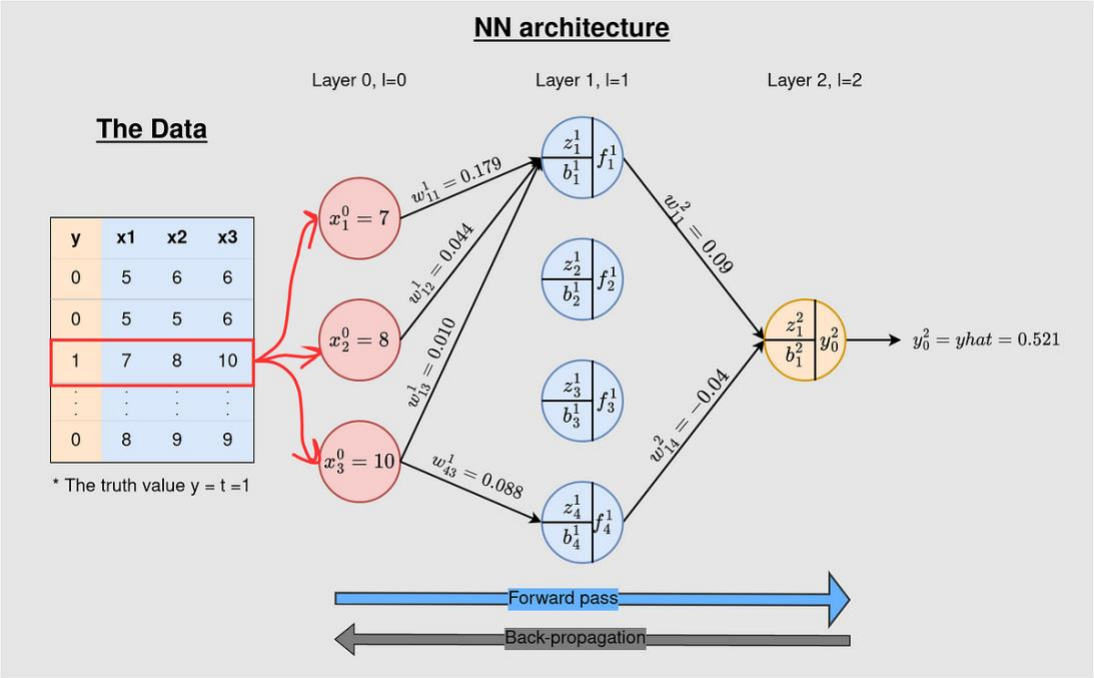


Backpropagation is a fundamental algorithm for training artificial neural networks, and it operates on the principle of efficiently computing gradients for each of the network's parameters (i.e., connection weights and biases) to minimize error. Here's a breakdown of how backpropagation works, simplified into its key steps:

1. **Forward Pass**:
    - The algorithm processes input data in mini-batches, passing it through the network layer by layer from the input layer to the output layer. This stage is called the forward pass.
    - During the forward pass, the network computes the output for each layer until it reaches the final output layer. The output at each layer is based on the inputs it receives, the connection weights, and the biases, using the layer's activation function.
    - All intermediate outputs are saved for use in the backward pass.

2. **Computing the Error**:
    - The algorithm evaluates the network's output against the desired output (targets) using a loss function. This function quantifies how much error the network's predictions have compared to the actual targets.

3. **Backward Pass (Backpropagation)**:
    - Starting from the output layer, the algorithm calculates the gradient of the error with respect to each weight and bias in the network. This involves figuring out how much each parameter contributed to the overall error.
    - It accomplishes this by applying the chain rule of calculus, which allows it to efficiently propagate the error backwards through the network, layer by layer. This step determines the error gradient for every single model parameter.
    - This backward pass is where the term "backpropagation" comes from, as the error information propagates from the output layer back to the input layer.



4. **Gradient Descent**:
    - With the gradients computed, the algorithm then adjusts the weights and biases in the direction that minimally reduces the error, using gradient descent. This step involves a slight adjustment of each parameter opposite to its gradient.
    - The size of the adjustment is determined by the learning rate, a hyperparameter that controls how big a step the algorithm takes during gradient descent.

5. **Repetition**:
    - This process (forward pass, computing error, backward pass, gradient descent) is repeated for each mini-batch of the training data, and the whole training dataset is passed through the network multiple times (each pass is called an epoch). With each epoch, the network parameters are adjusted to minimize the error further.

Automatic Differentiation (Autodiff):

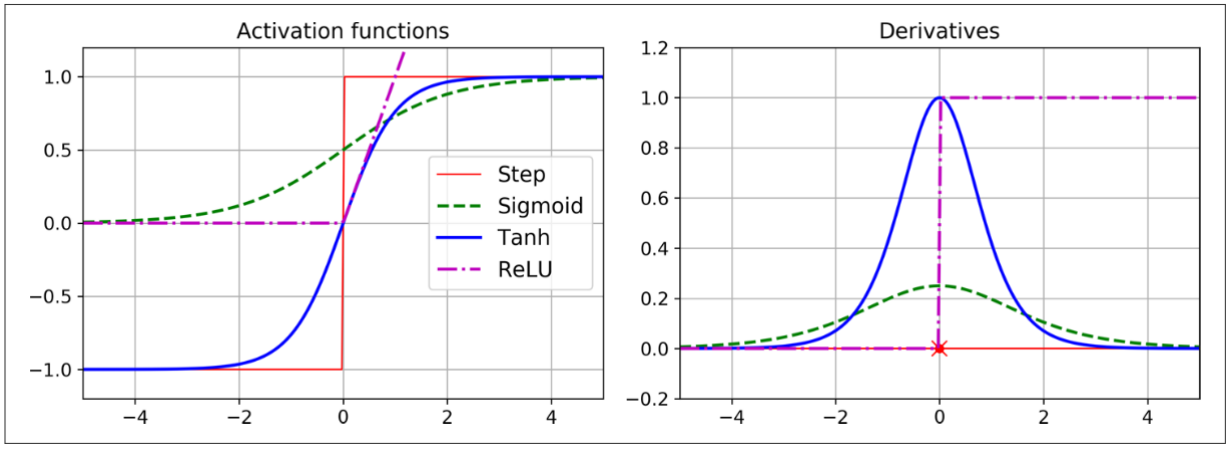
Backpropagation uses a technique called automatic differentiation, particularly reverse-mode autodiff, to compute these gradients efficiently. This method is especially effective when the function to be differentiated (the network's error function, in this case) has many inputs (weights and biases) and few outputs (the loss).

By iteratively adjusting the network's weights and biases to minimize the loss function, backpropagation enables the neural network to learn from the training data, improving its accuracy over time. This process is crucial for the development of neural networks that can perform complex tasks with high levels of precision.
Also another thing to take into account is that for using Gradient Descend we need to change the step function to the logistic function (sigmoid), because if it doesn't have a gradient and is flat we can't do the Gradient Descend.

Here are other activation funcitons:

1. *The hyperbolic tangent function: tanh(z) = 2σ(2z) – 1*
   Just like the logistic function, this activation function is *S*-shaped, continuous, and differentiable, but its output value ranges from –1 to 1 (instead of 0 to 1 in the case of the logistic function). That range tends to make each layer's output more or less centered around 0 at the beginning of training, which often helps speed up convergence.

2. *The Rectified Linear Unit function: ReLU(z) = max(0, z)*
   The ReLU function is continuous but unfortunately not differentiable at $z = 0$ (the slope changes abruptly, which can make Gradient Descent bounce around), and its derivative is 0 for $z < 0$. In practice, however, it works very well and has the advantage of being fast to compute, so it has become the default. Most importantly, the fact that it does not have a maximum output value helps reduce some issues during Gradient Descent .



Regression & Classification MLPs

The number of outputs neurons will be the number of items you need, For example, to locate the center of an object in an image, you need to predict 2D coordinates, so you need two output neurons.

The loss function to use during training is typically the mean squared error, but if you have a lot of outliers in the training set, you may prefer to use the mean absolute error instead. Alternatively, you can use the Huber loss, which is a combination of both.

Hyperparameter	Typical value
# input neurons	One per input feature (e.g., 28 x 28 = 784 for MNIST)
# hidden layers	Depends on the problem, but typically 1 to 5
# neurons per hidden layer	Depends on the problem, but typically 10 to 100
# output neurons	1 per prediction dimension
Hidden activation	ReLU (or SELU, see Chapter 11)
Output activation	None, or ReLU/softplus (if positive outputs) or logistic/tanh (if bounded outputs)
Loss function	MSE or MAE/Huber (if outliers)

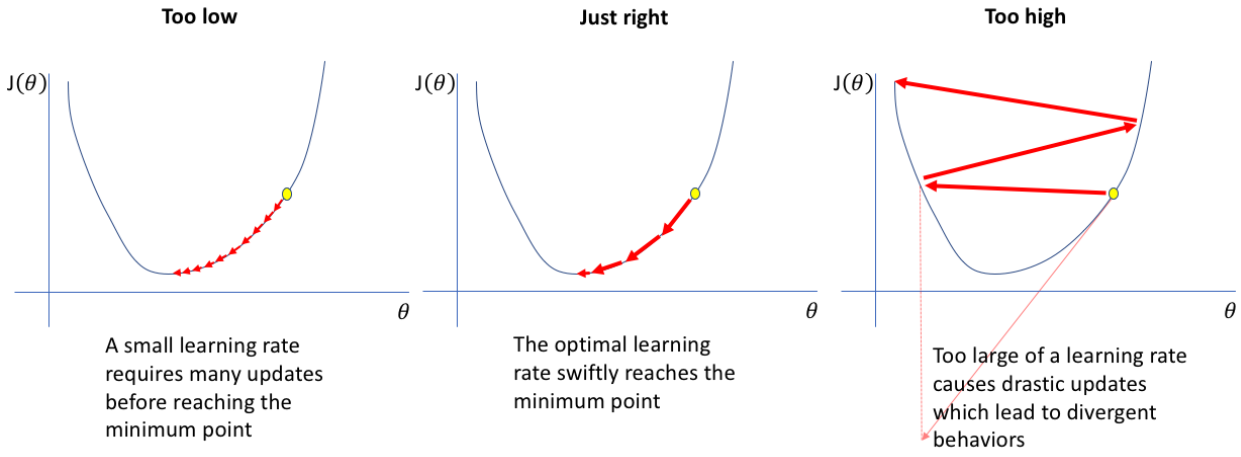
Hyperparameter	Binary classification	Multilabel binary classification	Multiclass classification
Input and hidden layers	Same as regression	Same as regression	Same as regression
# output neurons	1	1 per label	1 per class
Output layer activation	Logistic	Logistic	Softmax
Loss function	Cross entropy	Cross entropy	Cross entropy

General rules of thumb when building models:

1. **Number of Hidden Layers**: Start simple with one or two hidden layers. As the task becomes more complex, increase the number of layers until you start overfitting.

2. **Number of Neurons per Layer**: You can start with a number of neurons in the same way as the number of inputs. If the model is overfitting, you might need to reduce the number, or if underfitting, try increasing it. Using the same number of neurons in hidden layers, as a default, can simplify your model tuning.

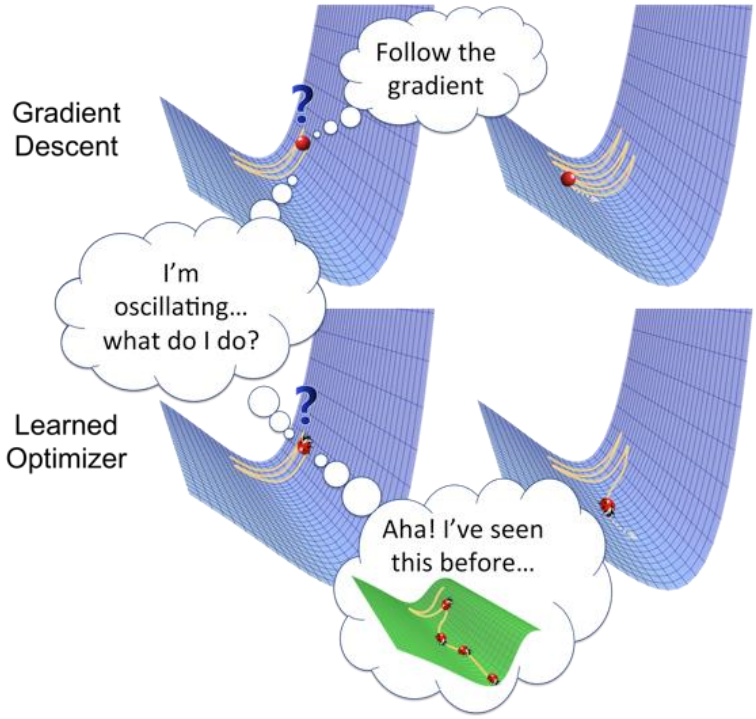
3. **Learning Rate**: It's often the most crucial hyperparameter. If it's too high, the model may fail to converge or even diverge. If it's too low, convergence will be slow, and the model might get stuck in suboptimal solutions. The right learning rate achieves efficient training without overshooting the loss function's minimum.



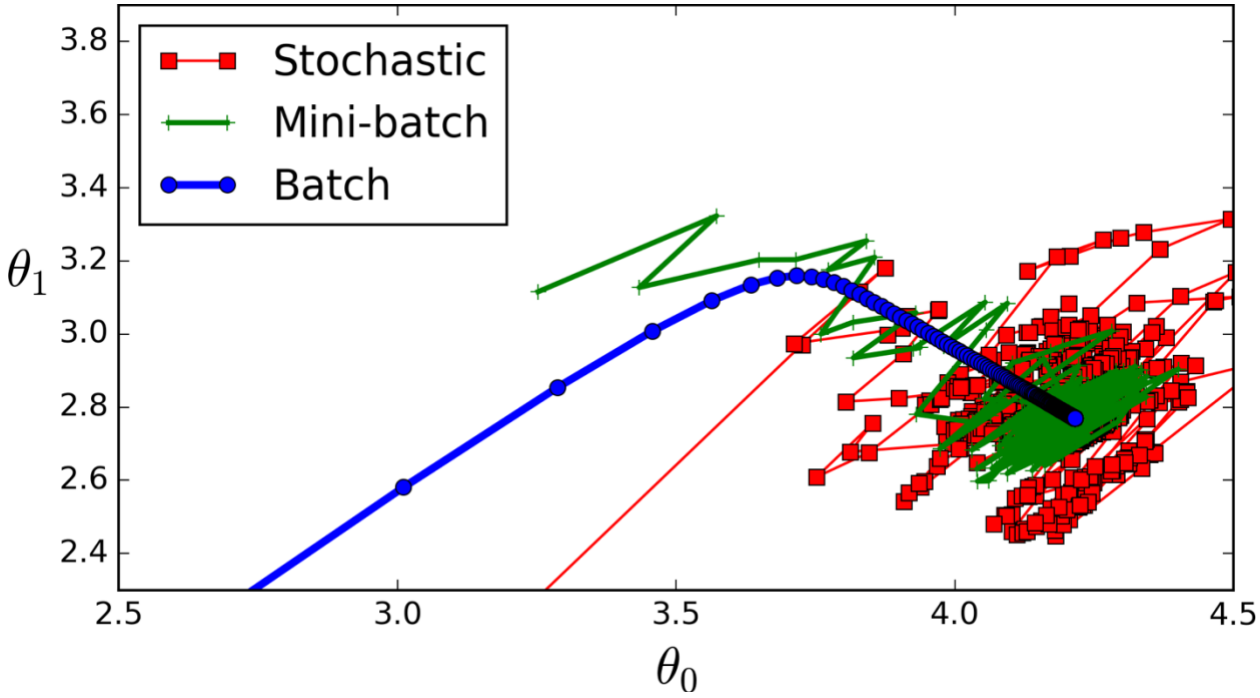
Learning rate schedules can adjust the learning rate during training to overcome the limitations of a fixed learning rate:

- **Time-Based Decay**: Reduces the learning rate gradually over time.
- **Step Decay**: Decreases the learning rate at specific epochs.
- **Exponential Decay**: Rapidly decreases the learning rate exponentially.
- **Learning Rate Warmup**: Starts with a low learning rate and increases it gradually to stabilize training.
- **Adaptive Methods**: Algorithms like AdaGrad, RMSprop, or Adam adjust the learning rate based on the gradients' behavior, offering a tailored approach for each parameter.

4. **Optimizer**: Optimizers like Adam, Nadam, or RMSprop often outperform SGD but try various options and see which works best for your specific problem.

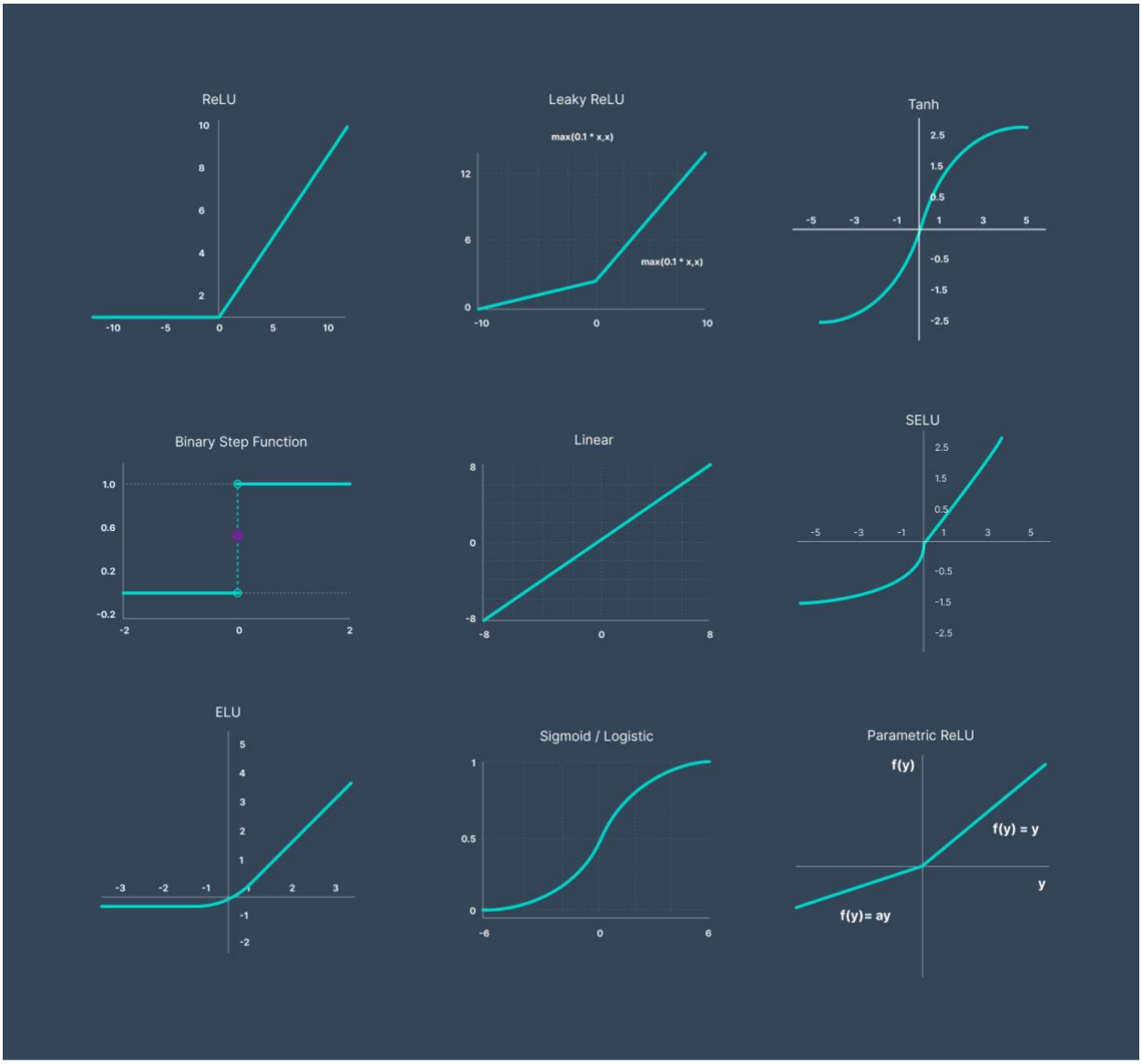


5. **Batch Size**: Larder batches provide a faster computation at the cost of a potentially less stable convergence. You may start with small batches for more robust convergence and experiment with larger sizes for faster computation.



Batch Gradient Descent takes the most direct path but is slow, whereas SGD is faster but much noisier. Mini-batch tries to balance both aspects, offering a middle ground between the computational efficiency of Batch and the faster convergence of SGD.

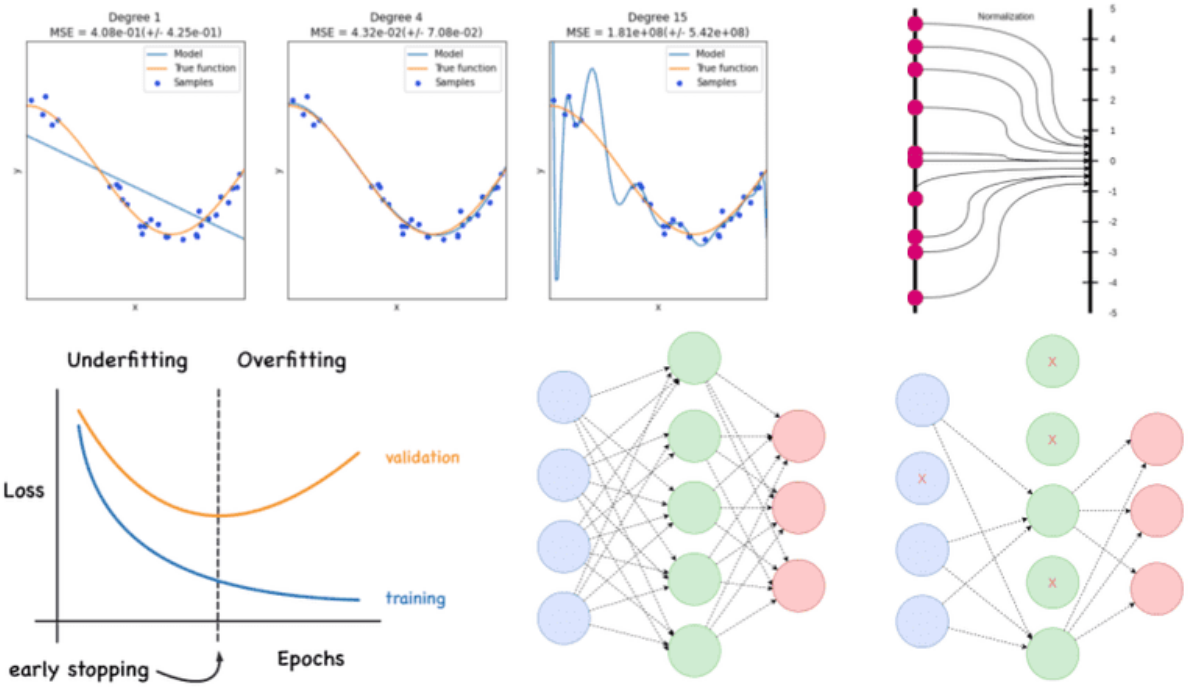
6. **Activation Functions**: Use ReLU (or its variants like Leaky ReLU or ELU) for hidden layers. For the output layer, the choice depends on your task (e.g., softmax for multi-class classification, sigmoid for binary classification, no activation for regression).



7. **Regularization**: Regularization techniques are key to preventing neural networks from overfitting:

    - **Dropout**: Temporarily removes a random subset of neurons in each training step to make the network less sensitive to specific weights.
    - **L1 Regularization**: Pushes weights towards zero, creating a sparse network and potentially eliminating unimportant features.
    - **L2 Regularization**: Penalizes large weights, spreading out importance and reducing overreliance on particular neurons.
    - **Early Stopping**: Halts training when performance on a validation set starts to worsen, avoiding overtraining.

    Heuristic approaches for applying regularization involve initially establishing a baseline without it, then incrementally adding techniques like dropout, and tuning regularization hyperparameters based on validation set performance. The aim is to apply just enough regularization to reduce overfitting without significantly hindering the network's learning capacity.



8. **Epochs and Early Stopping**: Rather than setting a fixed number of epochs, use early stopping to terminate training once the model stops improving on a validation set.

9. **Initialization**: The way weights are initially set can significantly affect how quickly the network learns and converges to a solution:

    - **He Initialization**: Particularly suited for layers with ReLU activation functions, this strategy initializes weights based on the number of input neurons, helping to prevent vanishing or exploding gradients in deep networks.
    - **Glorot Initialization**: Also known as Xavier initialization, this is a good all-around choice for layers with Sigmoid or Tanh activation functions, setting weights based on the average number of input and output neurons, aiming to maintain the variance of activations across layers.

10. **Data Preprocessing**: Always standardize your input features. For image data, use data augmentation to prevent overfitting.
11. **Evaluation**: Use a separate validation set to monitor for overfitting during training and a test set for final evaluation.
12. **Loss Function**: Choose the loss function according to your specific problem (e.g., cross-entropy for classification, mean squared error for regression).