Neural Networks, Gradient Descent, Backpropagation

Hersh Kumar

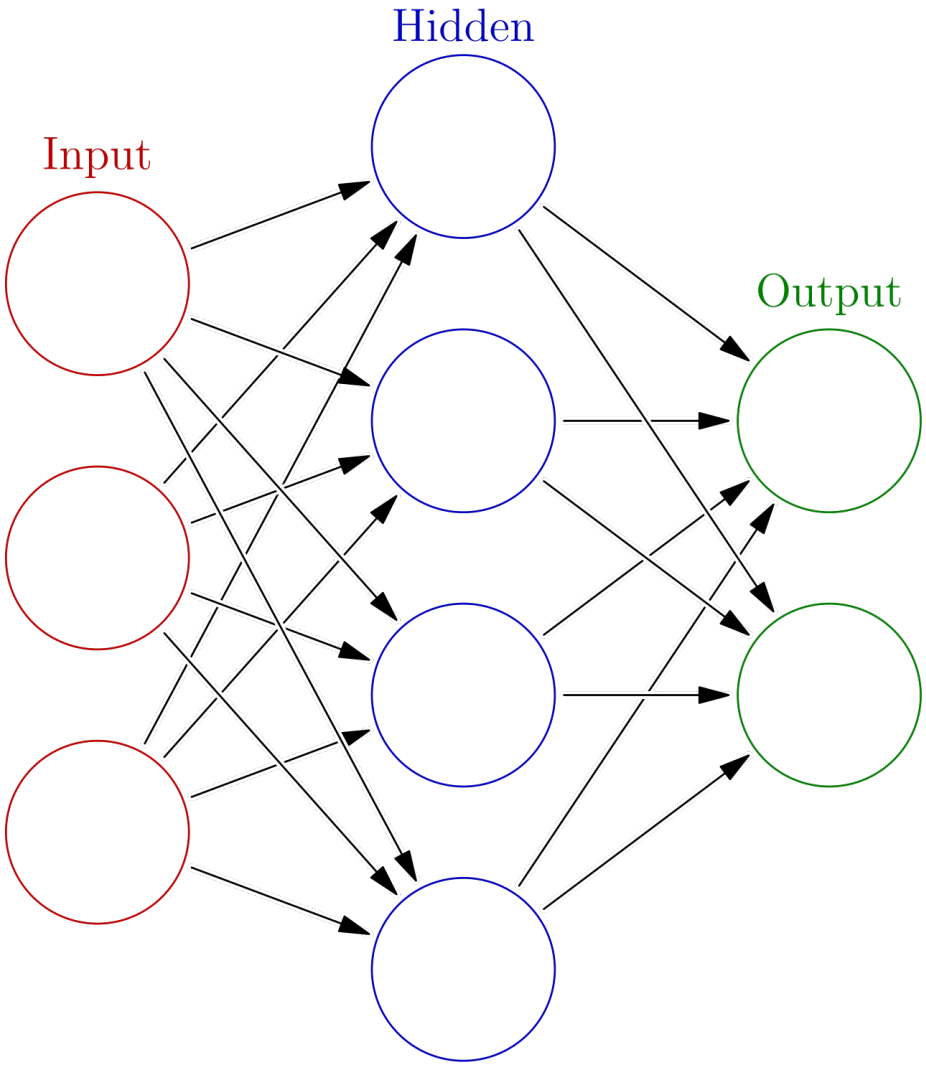
University of Maryland, Department of Physics

1. Neural Networks
2. Gradient Descent
3. Backpropagation
4. Training Neural Networks

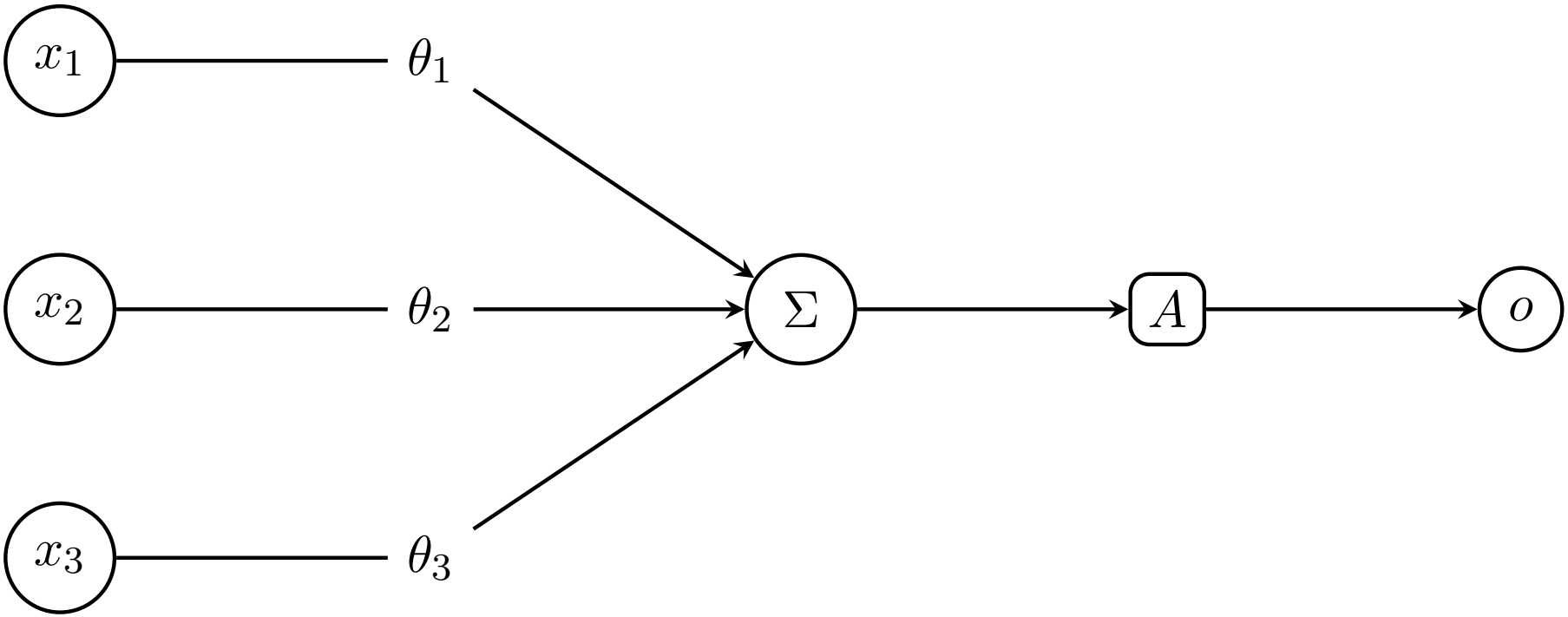
Neural Networks

- Consist of nodes (neurons) connected by directed edges
- Each node takes in the inputs of its preceding nodes, and applies some nonlinear activation function to the sum of the inputs
- The incoming connections are each scaled by a weight
- Many different types of networks:
  - Feedforward (MLP)
  - Convolutional NN (2D input processing)
  - Recurrent NN (Information flows forwards and backwards)
  - Many more variations...
- Weights are stored in matrices, evaluating the neural network is done via matrix multiplication $\rightarrow$ good on GPUs

Feedforward Neural Network



Single Node Schematic



How do we train neural networks to be useful?

First we need to quantify the usefulness of the network:

How do we train neural networks to be useful?

First we need to quantify the usefulness of the network:

- Given some network N, parameterized by $\boldsymbol{\theta}$, we can measure how "good" the network is at its purpose.
- To quantify this, we use a loss function, which is problem specific:
  - For classification tasks, we can consider the prediction confidence (log loss/cross entropy)
  - If the network is producing a distribution of values, we can consider the KL divergence

How do we train neural networks to be useful?

First we need to quantify the usefulness of the network:

- Given some network N, parameterized by $\boldsymbol{\theta}$, we can measure how "good" the network is at its purpose.
- To quantify this, we use a loss function, which is problem specific:
  - For classification tasks, we can consider the prediction confidence (log loss/cross entropy)
  - If the network is producing a distribution of values, we can consider the KL divergence

  We will want to vary the parameters $\boldsymbol{\theta}$ in order to minimize the loss function.

Gradient Descent

- In the space of parameter configurations, there is some ideal $\boldsymbol{\theta}$ such that the loss function is minimized $\rightarrow$ Network does the best job possible[1].
- The question is, how do we find the ideal set of parameters?
- Looking at the function $f(\boldsymbol{\theta})$, $\nabla f(\boldsymbol{\theta})$ at every point points in the direction along the maximal increase
- $-\nabla f(\boldsymbol{\theta})$ will point along the direction of maximal **decrease**
  - Since f must be differentiable, we can locally approximate it as a plane
  - Direction opposite the maximal increase direction on a plane is the direction of maximal decrease

[1] Given the architecture.

Gradient Descent

- In the space of parameter configurations, there is some ideal $\boldsymbol{\theta}$ such that the loss function is minimized $\rightarrow$ Network does the best job possible[1].
- The question is, how do we find the ideal set of parameters?
- Looking at the function $f(\boldsymbol{\theta})$, $\nabla f(\boldsymbol{\theta})$ at every point points in the direction along the maximal increase
- $-\nabla f(\boldsymbol{\theta})$ will point along the direction of maximal **decrease**
  - Since f must be differentiable, we can locally approximate it as a plane
  - Direction opposite the maximal increase direction on a plane is the direction of maximal decrease
- Moving **against** the gradient minimizes the function $f(\boldsymbol{\theta})$

[1] Given the architecture.

Pseudocode

Algorithm 1 Naive Gradient Descent

Require: Initial parameters θ_0, learning rate α, tolerance ε

Ensure: Optimized parameters θ

1: $\theta = \theta_0$
2: **repeat**
3: Compute gradient: $g = \nabla f(\theta)$
4: Update parameters: $\theta = \theta - \alpha\, g$
5: **until** $\|g\| < \varepsilon$
6: **return** θ

Caveat: Gradient Descent does not ensure global minima, just local minima (and only if you pick the hyperparameters correctly)

Other Gradient Descent Algorithms

Optimizers

`adabelief` (learning_rate[, b1, b2, eps, ...])	The AdaBelief optimizer.
`adadelta` ([learning_rate, rho, eps, ...])	The Adadelta optimizer.
`adan` (learning_rate[, b1, b2, b3, eps, ...])	The ADAptive Nesterov momentum algorithm (Adan).
`adafactor` ([learning_rate, ...])	The Adafactor optimizer.
`adagrad` (learning_rate[, ...])	The Adagrad optimizer.
`adam` (learning_rate[, b1, b2, eps, eps_root, ...])	The Adam optimizer.
`adamw` (learning_rate[, b1, b2, eps, ...])	Adam with weight decay regularization.
`adamax` (learning_rate[, b1, b2, eps])	A variant of the Adam optimizer that uses the infinity norm.
`adamaxw` (learning_rate[, b1, b2, eps, ...])	Adamax with weight decay regularization.
`amsgrad` (learning_rate[, b1, b2, eps, ...])	The AMSGrad optimizer.
`fromage` (learning_rate[, min_norm])	The Frobenius matched gradient descent (Fromage) optimizer.
`lamb` (learning_rate[, b1, b2, eps, eps_root, ...])	The LAMB optimizer.
`lars` (learning_rate[, weight_decay, ...])	The LARS optimizer.
`lbfgs` ([learning_rate, memory_size, ...])	L-BFGS optimizer.
`lion` (learning_rate[, b1, b2, mu_dtype, ...])	The Lion optimizer.
`nadam` (learning_rate[, b1, b2, eps, ...])	The NAdam optimizer.
`nadamw` (learning_rate[, b1, b2, eps, ...])	NAdamW optimizer, implemented as part of the AdamW optimizer.
`noisy_sgd` (learning_rate[, eta, gamma, key, seed])	A variant of SGD with added noise.

`fromage` (learning_rate[, min_norm])	The Frobenius matched gradient descent (Fromage) optimizer.
`lamb` (learning_rate[, b1, b2, eps, eps_root, ...])	The LAMB optimizer.
`lars` (learning_rate[, weight_decay, ...])	The LARS optimizer.
`lbfgs` ([learning_rate, memory_size, ...])	L-BFGS optimizer.
`lion` (learning_rate[, b1, b2, mu_dtype, ...])	The Lion optimizer.
`nadam` (learning_rate[, b1, b2, eps, ...])	The NAdam optimizer.
`nadamw` (learning_rate[, b1, b2, eps, ...])	NAdamW optimizer, implemented as part of the AdamW optimizer.
`noisy_sgd` (learning_rate[, eta, gamma, key, seed])	A variant of SGD with added noise.

`novograd` (learning_rate[, b1, b2, eps, ...])	NovoGrad optimizer.
`optimistic_gradient_descent` (learning_rate[, ...])	An Optimistic Gradient Descent optimizer.
`optimistic_adam` (learning_rate[, optimism, ...])	The Optimistic Adam optimizer.
`polyak_sgd` ([max_learning_rate, scaling, ...])	SGD with Polyak step-size.
`radam` (learning_rate[, b1, b2, eps, ...])	The Rectified Adam optimizer.
`rmsprop` (learning_rate[, decay, eps, ...])	A flexible RMSProp optimizer.
`sgd` (learning_rate[, momentum, nesterov, ...])	A canonical Stochastic Gradient Descent optimizer.
`sign_sgd` (learning_rate)	A variant of SGD using only the signs of the gradient components.
`sm3` (learning_rate[, momentum])	The SM3 optimizer.
`yogi` (learning_rate[, b1, b2, eps])	The Yogi optimizer.

Adam Optimizer

- One of the "go-to" gradient descent algorithms
- Implements the benefits of RMSProp and Momentum
  - RMSProp deals with different parameters having different magnitudes of derivatives
  - Momentum takes into account the previous gradients, and increases the scale of the gradient based on if the gradient has been moving in the same direction
- Adds three hyperparameters, β_1, β_2, and ε, which are generally left at default values

Computing $\nabla f(\theta)$

- The key task of the gradient descent algorithm is the computation of the gradient in parameter space of the loss function
- To compute the gradient in parameter space of a neural network, we apply backpropagation
- Backpropagation allows for layer-by-layer computation of the dependence of the loss function on each parameter in the network
- Generally done under the hood by ML packages

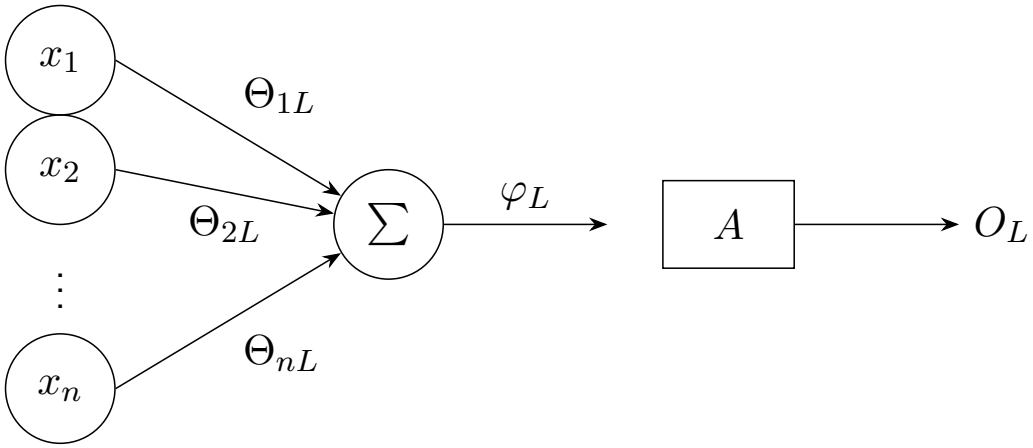
Backpropagation

- Naive method to estimate the gradient is finite difference
  - for each parameter, evaluate f, shift the parameter slightly, reevaluate f
  - Number of calls to neural network scales with number of parameters
- Backpropagation utilizes the chain rule and allows for efficient reuse of computation, number of total calls to neural network is constant

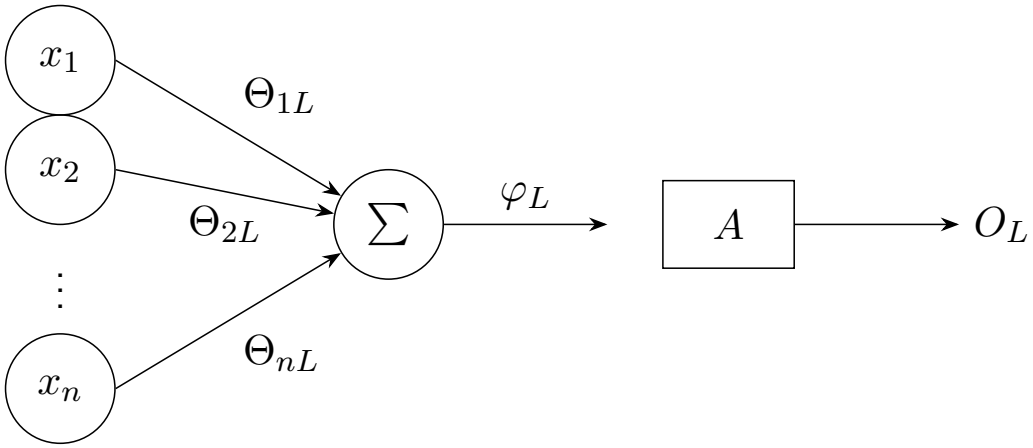
Backpropagation

- Given a layer L, we can determine the partial derivatives of the output of nodes in L with respect to the nodes in layer $L-1$
- We can then repeat this process all the way back to the input layer, and then use the chain rule to compute the partials of the loss function with respect to every parameter
- This is why differentiable activation functions are generally required $\rightarrow$ Need to be able to apply the chain rule on their inputs!

Single Layer Example



Single Layer Example



$$\frac{\partial f}{\partial \Theta_{iL}} = \frac{\partial f}{\partial O_L} \frac{\partial A}{\partial \varphi_L} \frac{\partial \varphi_L}{\partial \Theta_{iL}}$$

$$\frac{\partial \varphi_L}{\partial \Theta_{iL}} = \frac{\partial}{\partial \Theta_{iL}} \left[\sum_{k=1}^{n} \Theta_{kL} x_k \right] = \frac{\partial}{\partial \Theta_{iL}} \Theta_{iL} x_i = x_i$$

Backpropagation

- The remaining derivative in the chain rule expansion, $\frac{\partial f}{\partial O_L}$, is more difficult to compute for internal layers, but there is a recursive way to compute it[2]:

$$\frac{\partial f}{\partial O_L} = \sum_k \left(\frac{\partial f}{\partial O_k} \frac{\partial O_k}{\partial \varphi_k} \Theta_{kL} \right)$$

- All together, we can compute $\frac{\partial f}{\partial \Theta_{ij}}$, the gradient with respect to any parameter in any layer of the network

[2] For a much more complete explanation, with worked examples, see this lecture.

Backpropagation

- Backpropagation allows for the efficient computation of gradients when the number of parameters is large, unlike finite difference methods
- Can also be represented via a series of matrix operations, which allows for easier implementation (and GPU acceleration)
- Under the hood of backpropagation, automatic differentiation provides efficient calculation of the derivatives of the activation function.

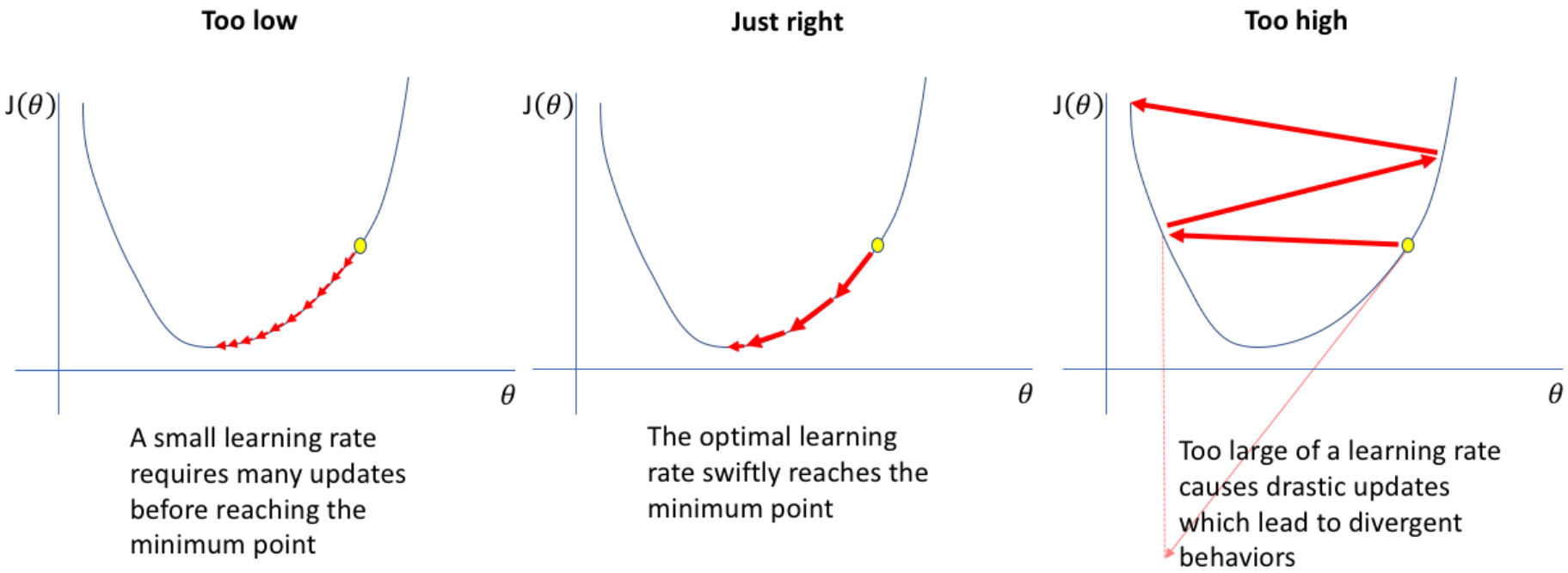
Training Overview

1. Initialize a neural network with some initial parameters $\boldsymbol{\theta}_0$.
2. Define a loss function $f(\boldsymbol{\theta})$ that is characteristic to the problem that is being solved by the network
3. Obtain a set of data for which the correct outputs are known, split into a training set and a validation set
4. Using gradient descent, obtain optimized parameters $\boldsymbol{\theta}_f$, by querying the network on the training dataset
5. After training, use the validation dataset (which the network has never seen), to evaluate the abilities of the network
6. Repeat steps 4 and 5 as needed

Hyperparameters

1. Learning Rate
   - Larger learning rates can be optimal for early training, the network is far from the ideal parameters
   - Smaller learning rates are ideal for later stages of training
2. Gradient Descent Iterations
   - Training for too long $\rightarrow$ overfitting
   - Dividing labeled data into training and validation sets
3. Problem-dependent parameters, such as the number of Monte Carlo samples, any other factors relevant for data generation

Choosing Learning Rate



Digit Classification

Given an image of a handwritten digit, 0-9, we want to determine the digit drawn.

- Problem Type: Multiclass classification
- Network Architecture: Multilayer Perceptrons, Convolutional Neural Networks
- Loss Function: Many (ex. Categorical Cross-Entropy, KL Divergence)
- Gradient Descent Algorithm: Many (Adam, Stochastic Gradient Descent, etc.)