**1. Neural Networks**

**1.1 Single-Layer Neural Networks**

A single-layer neural network (a.k.a. a *perceptron* or *logistic unit*, depending on activation) computes:

y=σ(wTx+b)\mathbf{y} = \sigma(\mathbf{w}^T \mathbf{x} + b)y=σ(wTx+b)

* x\mathbf{x}x : Input vector
* w\mathbf{w}w : Weight vector (model parameters)
* bbb : Bias (scalar)
* σ(⋅)\sigma(\cdot)σ(⋅) : Nonlinear activation function (e.g., Sigmoid, ReLU)

**Interpretation:**  
A single neuron effectively performs a linear combination of the inputs and then passes it through a (usually) nonlinear function. Historically, the sigmoid function was popular; modern practice often uses ReLU and its variants for better training behavior.

**What kind of decision functions are represented by a 1-layer NN?**  
A single-layer neural network typically represents a *linear decision boundary* (once the nonlinear activation is taken into account, the final effect is still a single line or hyperplane in the input space, especially if the activation is a step or sigmoid for classification).

**1.2 Multilayer Neural Networks**

A multilayer neural network (often called a *Deep Neural Network* if it has many layers) stacks these neurons in multiple layers:

y=σ(W(1) σ(W(0)x+b(0))+b(1))\mathbf{y} = \sigma\Bigl(W^{(1)} \,\sigma\bigl(W^{(0)} \mathbf{x} + \mathbf{b}^{(0)}\bigr) + \mathbf{b}^{(1)}\Bigr)y=σ(W(1)σ(W(0)x+b(0))+b(1))

* W(i)W^{(i)}W(i) : Weights for layer iii
* b(i)\mathbf{b}^{(i)}b(i) : Bias for layer iii
* σ\sigmaσ : Activation function, applied element-wise

**Universal Approximation:**  
A two-layer network with enough hidden units can approximate *any* continuous function on Rn\mathbb{R}^nRn. This is called the *Universal Approximation Theorem*.

**Training / Optimization:**  
We typically train neural networks by defining a *loss function* (e.g., mean squared error for regression, cross-entropy for classification) and then using *gradient descent* or a variant to update parameters (w,b\mathbf{w}, \mathbf{b}w,b).

**2. Deep Learning in Keras**

Keras (with TensorFlow backend) is a popular high-level library for creating and training neural networks.

**2.1 Basic Workflow in Keras**

1. **Create a model (usually Sequential)**

python

Copy code

from tensorflow.keras.models import Sequential

model = Sequential()

1. **Add layers**

python

Copy code

from tensorflow.keras.layers import Dense, Activation

model.add(Dense(units=64, input\_dim=100, activation='relu'))

model.add(Dense(units=10, activation='softmax'))

1. **Compile the model**

python

Copy code

model.compile(loss='categorical\_crossentropy',

optimizer='sgd',

metrics=['accuracy'])

You can customize the optimizer further, e.g.:

python

Copy code

from tensorflow.keras.optimizers import SGD

model.compile(loss='categorical\_crossentropy',

optimizer=SGD(lr=0.01, momentum=0.9, nesterov=True),

metrics=['accuracy'])

1. **Train (fit) the model**

python

Copy code

model.fit(X\_train, Y\_train, epochs=5, batch\_size=32)

1. **Evaluate the model**

python

Copy code

loss\_and\_metrics = model.evaluate(X\_test, Y\_test, batch\_size=32)

1. **Predict using the model**

python

Copy code

classes = model.predict\_classes(X\_test, batch\_size=32)

proba = model.predict\_proba(X\_test, batch\_size=32)

**2.2 Example: MNIST Digit Classification**

1. **Data loading and preparation**
   * MNIST data: 60,000 training images, 10,000 test images of handwritten digits (0–9).
   * Reshape each 28×2828 \times 2828×28 image into a 784-dimensional vector.
   * Normalize pixel values to [0,1][0, 1][0,1] by dividing by 255.
   * Convert labels to one-hot encodings (e.g., 2 →\to→ [0, 0, 1, 0, ..., 0]).
2. **Model architecture**

python

Copy code

model = Sequential()

model.add(Dense(16, activation='relu', input\_shape=(784,)))

model.add(Dropout(0.2))

model.add(Dense(32, activation='relu'))

model.add(Dropout(0.2))

model.add(Dense(num\_classes, activation='softmax'))

1. **Compile the model** with a chosen optimizer, e.g. Adagrad, RMSprop, Adam, etc.

python

Copy code

from tensorflow.keras.optimizers import Adagrad

my\_opt = Adagrad(learning\_rate=0.01, epsilon=0.1)

model.compile(loss='categorical\_crossentropy',

optimizer=my\_opt,

metrics=['accuracy'])

1. **Train**

python

Copy code

history = model.fit(x\_train, y\_train,

batch\_size=64,

epochs=20,

validation\_data=(x\_test, y\_test))

1. **Evaluate**

python

Copy code

score = model.evaluate(x\_test, y\_test, verbose=0)

print('Test loss:', score[0])

print('Test accuracy:', score[1])

**2.3 Dropout**

* **Motivation:** Prevent overfitting by randomly “dropping out” (setting to zero) certain neuron outputs during training.
* **Effect:** Forces the network to learn more robust features, because it cannot rely on specific neurons always “co-firing.”

python

Copy code

from tensorflow.keras.layers import Dropout

model.add(Dense(64, activation='relu'))

model.add(Dropout(0.5))

**2.4 Keras Optimizers**

Keras provides multiple optimizers that are variants of gradient descent:

1. **SGD** (Stochastic Gradient Descent)
2. **RMSprop**
3. **Adagrad**
4. **Adadelta**
5. **Adam**
6. **Adamax**
7. **Nadam**

**Example instantiation**:

python

Copy code

from tensorflow.keras.optimizers import SGD, RMSprop, Adagrad, Adam, ...

my\_opt = SGD(lr=0.01, momentum=0.9, decay=0.0, nesterov=True)

**Key Ideas Behind Some Optimizers**

* **Momentum (used by SGD)**  
  Helps accelerate gradients in the right direction, dampening oscillations.

vt=m vt−1+α ∇wf,w=w−vtv\_t = m \, v\_{t-1} + \alpha \, \nabla\_w f, \quad w = w - v\_tvt​=mvt−1​+α∇w​f,w=w−vt​

Where mmm is the momentum constant (e.g. 0.9), α\alphaα is the learning rate.

* **Adagrad**  
  Adapts the learning rate for each parameter based on how frequently it’s updated. Parameters that get **frequent** updates get **smaller** learning rates; those with **infrequent** updates get **larger** learning rates.

c=c+(∇wf)2,w=w−αcc = c + (\nabla\_w f)^2, \quad w = w - \frac{\alpha}{\sqrt{c}}c=c+(∇w​f)2,w=w−c​α​

* **Adam**  
  Combines ideas from RMSprop (adaptive learning rate) and Momentum. Usually a good default choice when starting out.

**Key Takeaways**

1. **Single-layer networks** produce linear decision boundaries (when used for classification with a single output).
2. **Multilayer networks** stack layers of neurons and can approximate virtually any continuous function (universal approximation).
3. **Gradient descent** (and its variants) is the core method to train neural networks via *backpropagation*.
4. **Keras** streamlines the process of building, training, and evaluating neural networks with intuitive APIs.
5. **Dropout** helps combat overfitting by randomly zeroing out neuron outputs during training.
6. **Optimizers** in Keras (SGD, RMSprop, Adagrad, Adam, etc.) enable fine-grained control over how gradients are used for updates.

## 2. Deep Learning

## 0. Before We Start

* Simple methods for numerical optimization (like **Nelder-Mead**) do not use gradients; instead, they **sample** points and adapt step sizes depending on whether they find better or worse points.
* For high-dimensional problems, sampling-based approaches become very expensive.
* **Gradient-based** methods typically scale better in high dimensions and are thus preferred in many machine learning applications.

**Key Questions:**

1. What are the computational limitations of Nelder-Mead or other sampling-based methods?
   * They require many function evaluations, grow in complexity with dimensionality, and do not exploit gradient information.
2. When is it a real alternative?
   * When the function is not easily differentiable or is relatively low-dimensional, or when the gradient is expensive or undefined.

## 1. Gradient Descent

### 1.1 The One-Dimensional Case

Given f(w):R→Rf(w): \mathbb{R} \to \mathbb{R}f(w):R→R, we want the www that minimizes fff.

* **Derivative** f′(w)f'(w)f′(w) is the rate of change of fff at www.
* **Idea:** Update www by stepping in the negative direction of the derivative: wnew=w−α f′(w). w^{\text{new}} = w - \alpha \, f'(w).wnew=w−αf′(w).
  + α\alphaα is the learning rate or step size.
  + If α\alphaα is too large, you may overshoot. If too small, convergence is slow.

**Critical Points**:

* If f′(w)=0f'(w) = 0f′(w)=0, www can be a minimum, a maximum, or a saddle point.
* If fff is convex, a derivative of 0 implies a global minimum.

**Numerical Derivatives**:

* Approximate f′(x)≈f(x+h)−f(x−h)2hf'(x)\approx \frac{f(x+h)-f(x-h)}{2h}f′(x)≈2hf(x+h)−f(x−h)​.
* This is often slow and introduces numerical errors.

**Analytical Derivatives**:

* If we know a closed-form for f′(w)f'(w)f′(w), we can compute it directly.
* This avoids the cost and errors of numerical differentiation.

## 2. From Derivatives to Gradient in nnn-Dimensions

* We generalize from ddw\frac{d}{dw}dwd​ to the **gradient** ∇f\nabla f∇f, where ∇f=(∂f∂x1,…,∂f∂xn). \nabla f = \left(\frac{\partial f}{\partial x\_1}, \dots, \frac{\partial f}{\partial x\_n}\right).∇f=(∂x1​∂f​,…,∂xn​∂f​).
* In multiple dimensions, gradient descent updates the vector x\mathbf{x}x via xnew=x−α ∇f(x). \mathbf{x}^{\text{new}} = \mathbf{x} - \alpha \,\nabla f(\mathbf{x}).xnew=x−α∇f(x).
* **Cost of Finite Differences**: For an nnn-dimensional input, naive finite differences require 2n2n2n function evaluations just to compute ∇f\nabla f∇f. This can be extremely expensive.

## 3. How to Learn from Data?

### 3.1 The Machine Learning Perspective

* We have a dataset (x,y)(\mathbf{x}, y)(x,y). We want to find parameters w\mathbf{w}w of a model MMM that minimize a **loss function**: f(x,y)(w)=(loss between model predictions and the true labels). f\_{(\mathbf{x},y)}(\mathbf{w}) = \text{(loss between model predictions and the true labels)}.f(x,y)​(w)=(loss between model predictions and the true labels).

### Common Loss Functions

1. **Square (Euclidean) Loss** L=1n∑i(yi−f(xi))2. L = \frac{1}{n} \sum\_i (y\_i - f(\mathbf{x}\_i))^2.L=n1​i∑​(yi​−f(xi​))2.
2. **Hinge Loss** (SVM): L=1n∑imax⁡(0,1−yif(xi)). L = \frac{1}{n} \sum\_i \max(0, 1 - y\_i f(\mathbf{x}\_i)).L=n1​i∑​max(0,1−yi​f(xi​)).
3. **Logistic Loss** (Logistic Regression): L=1n∑ilog⁡(1+e−yif(xi)). L = \frac{1}{n} \sum\_i \log \bigl(1 + e^{-y\_i f(\mathbf{x}\_i)}\bigr).L=n1​i∑​log(1+e−yi​f(xi​)).
4. **Cross-Entropy Loss** + **Softmax** (multiclass problems): CrossEntropy(y,y^)=−∑jyjlog⁡y^j. \text{CrossEntropy}(y, \hat{y}) = -\sum\_j y\_j \log \hat{y}\_j.CrossEntropy(y,y^​)=−j∑​yj​logy^​j​. Where yyy is the “true” one-hot vector and y^\hat{y}y^​ is the predicted probability distribution.

## 4. Batch Gradient Descent vs. Stochastic Gradient Descent

### 4.1 Batch Gradient Descent (BGD)

* Uses all training samples to compute an **exact** gradient at each step:

python

Copy code

for epoch in range(num\_epochs):

grad = evaluate\_gradient\_on\_full\_dataset(data, w)

w = w - alpha \* grad

* This can be slow when the dataset is very large.

### Stochastic Gradient Descent (SGD)

* Updates parameters **per sample** (or per mini-batch), thus using a noisy approximation of the true gradient:

python

Copy code

for epoch in range(num\_epochs):

np.random.shuffle(data)

for sample in data: # one at a time

grad = evaluate\_gradient(sample, w)

w = w - alpha \* grad

* Usually converges faster in terms of wall-clock time when dataset size is large.
* The gradient is noisy but that noise can help escape local minima in non-convex problems.

### Mini-batch Gradient Descent

* Combines the best of both worlds: uses a small batch ≪\ll≪ full dataset but >1> 1>1.

python

Copy code

for epoch in range(num\_epochs):

np.random.shuffle(data)

for batch in get\_batches(data, batch\_size=50):

grad = evaluate\_gradient(batch, w)

w = w - alpha \* grad

* More stable than pure SGD (less noisy), yet still more efficient than full BGD.

## EXTRA 5. Automatic Differentiation

* **Manual Differentiation**: Tedious and error-prone for large models.
* **Finite Differences**: Approximate but slow (2n2n2n evaluations in nnn-dim).
* **Symbolic Differentiation**: Tools like Sympy can generate expressions but may become unwieldy for large models.
* **Automatic Differentiation** (AD): Systematically applies the chain rule at the operator level inside your program.
  + Splits into **Forward-Mode** AD (best for n≪mn \ll mn≪m) and **Reverse-Mode** AD (best for n≫mn \gg mn≫m, typical in neural nets).
  + **Backpropagation** is essentially reverse-mode AD applied to neural networks.

**Example**: 1-layer network

f(x)=11+e−(wTx+b).f(\mathbf{x}) = \frac{1}{1 + e^{- (\mathbf{w}^T\mathbf{x} + b) }}.f(x)=1+e−(wTx+b)1​.

* We can write fff as a sequence of elementary operations (multiply, add, exponentiate, etc.).
* Automatic differentiation tracks partial derivatives backward through these operations, yielding exact derivatives at machine precision.

**Conclusion**: AD is crucial in modern deep learning frameworks (e.g., TensorFlow, PyTorch). It frees us from manually coding derivatives and allows for efficient, large-scale gradient-based optimization.

# Key Takeaways

1. **Gradient Descent** relies on derivatives (or gradients) to navigate toward minima.
2. **Step Size (α\alphaα)** selection is crucial:
   * Too large: risk overshooting.
   * Too small: slow convergence.
   * Can be fixed or adaptively changed (learning rate scheduling).
3. **Stochastic (or Mini-batch) GD** often works better in large-scale machine learning because it updates more frequently and doesn’t need the full dataset at every step.
4. **Loss Functions** vary for regression (e.g., MSE) and classification (e.g., hinge, logistic, cross-entropy).
5. **Automatic Differentiation** (especially reverse-mode or backprop) is a cornerstone of modern deep learning libraries. It automates partial derivative computations at scale.