Deep Learning

Neural Networks, CNNs, RNNs, and Modern Techniques

The Big Picture

Deep learning is a subfield of machine learning focused on **neural networks** - computational models loosely inspired by biological neurons. After initial popularity in the 1980s and a period of decline, neural networks resurged after 2010 due to:

* **Massive computing power**: GPUs enable training of huge models
* **Big data**: Millions of labeled images, text documents, etc.
* **Algorithmic advances**: Better architectures, regularization, optimization

Key successes: Image classification (ImageNet), speech recognition (Siri, Alexa), machine translation (Google Translate), game playing (AlphaGo), and generative AI (ChatGPT, DALL-E).

1. Single Layer Neural Networks

1.1 The Basic Architecture

A neural network transforms inputs through layers of computation. The simplest network has three components:

Layer	Description	Example
Input Layer	Raw features X = (X1, ..., Xp)	784 pixels for MNIST digit
Hidden Layer	Nonlinear transformations of inputs	K = 256 hidden units
Output Layer	Final prediction using hidden activations	10 class probabilities

1.2 Computing Hidden Layer Activations

Each hidden unit k computes a **weighted sum** of inputs, then applies a **nonlinear activation function** g:

$$A_k = g\left(w_{k0} + \sum_{j=1}^{p} w_{kj} X_j\right)$$

Where:

* w_k0 is the bias term for unit k
* w_kj are the weights connecting input j to hidden unit k
* g(.) is the nonlinear activation function

Analogy: Think of each hidden unit as a 'feature detector'. It looks at a specific weighted combination of inputs and 'fires' (activates strongly) when it sees a pattern it's tuned to recognize.

1.3 Activation Functions

The activation function g introduces **nonlinearity**. Without it, stacking layers would just give another linear model!

1. Sigmoid (historically popular):

$$g(z) = \frac{1}{1+e^{-z}}$$

Squashes output to (0, 1). Problem: gradients vanish for very large/small inputs.

2. ReLU (Rectified Linear Unit - modern standard):

$$g(z) = (z)_+ = \max(0, z)$$

Returns z if positive, 0 otherwise. Simple, fast, and avoids vanishing gradients.

3. Tanh:

$$g(z) = \tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

Squashes to (-1, 1). Centered around zero, which can help training.

Activation	Range	Pros	Cons
Sigmoid	(0, 1)	Smooth, probabilistic	Vanishing gradients
ReLU	[0, inf)	Fast, no vanishing gradient	Dead neurons (if always negative)
Tanh	(-1, 1)	Zero-centered	Vanishing gradients

1.4 The Output Layer

For regression (predicting a continuous value):

$$\hat{Y} = \beta_0 + \sum_{k=1}^{K} \beta_k A_k$$

Simply a linear combination of hidden activations - like linear regression on learned features.

For classification (K classes):

Use the **softmax** function to convert raw scores Z_k into probabilities:

$$P(Y = k|X) = \frac{e^{Z_k}}{\sum_{l=1}^{K} e^{Z_l}}$$

* Outputs are always positive and sum to 1 (valid probability distribution)
* Larger Z_k means higher probability for class k
* The exponential amplifies differences between scores

Example: For digit recognition, if Z = [2.1, 0.3, 0.1, ...] for digits 0-9, softmax might output P = [0.75, 0.12, 0.08, ...], predicting digit '0' with 75% confidence.

1.5 Why Nonlinearity is Essential

Without activation functions, a neural network with any number of layers would collapse to a single linear transformation:

Linear(Linear(Linear(X))) = Linear(X)

The nonlinearity allows networks to learn **arbitrarily complex functions**. In fact, a neural network with one hidden layer and enough units can approximate any continuous function (Universal Approximation Theorem).

2. Multilayer (Deep) Neural Networks

2.1 Going Deeper

Modern networks stack multiple hidden layers. Each layer learns increasingly abstract representations:

* **Layer 1**: Learns simple features (edges, colors)
* **Layer 2**: Combines layer 1 features into parts (eyes, wheels)
* **Layer 3**: Combines parts into objects (faces, cars)
* **Deeper layers**: Even more abstract concepts

Example - Face Recognition: Early layers detect edges and textures. Middle layers detect eyes, noses, mouths. Deep layers recognize 'this combination of features = face of person X'.

2.2 Loss Functions

We train the network by minimizing a **loss function** that measures prediction error:

For regression - Squared Error Loss:

$$L = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

For classification - Cross-Entropy Loss:

$$L = -\sum_{i=1}^{n} \sum_{k=1}^{K} y_{ik} \log(p_{ik})$$

where y_ik = 1 if observation i belongs to class k, and p_ik is the predicted probability.

Intuition for cross-entropy: It penalizes confident wrong predictions heavily. If true class is k and we predict P(k) = 0.01, loss is -log(0.01) = 4.6. If we predict P(k) = 0.99, loss is only -log(0.99) = 0.01.

2.3 Overfitting and Regularization

Deep networks often have **millions of parameters** but may train on only thousands of examples. Overfitting is a serious concern!

Example: A network for MNIST might have 235,146 parameters but only 60,000 training images. It could easily memorize the training set.

Regularization methods:

1. Ridge (L2) Regularization:

$$L_{ridge} = L + \lambda \sum_{l} \sum_{j} w_{lj}^2$$

Penalizes large weights, encouraging simpler models.

2. Dropout:

During training, randomly 'drop' (set to zero) a fraction of neurons in each layer. This prevents neurons from co-adapting too much and forces the network to learn redundant representations.

Analogy for Dropout: Imagine a team where each member might randomly be absent. The team learns to not rely too heavily on any single member, making it more robust overall.

3. Early Stopping:

Monitor validation error during training. Stop when validation error starts increasing (even if training error is still decreasing).

3. Convolutional Neural Networks (CNNs)

3.1 The Problem with Fully Connected Networks for Images

A 256x256 color image has 256 x 256 x 3 = 196,608 input features. If the first hidden layer has 1000 units, that's ~197 million weights in just the first layer!

Problems:

* Too many parameters - easy to overfit
* Ignores spatial structure - nearby pixels are related
* Not translation invariant - cat in corner vs. center looks completely different

3.2 The Convolution Operation

Instead of connecting every input to every hidden unit, use small **filters** (kernels) that slide across the image:

$$A_{ij}^{(l)} = g\left(\sum_{s,t} K_{st} \cdot X_{i+s, j+t} + b\right)$$

Where:

* K is a small filter (e.g., 3x3 or 5x5 pixels)
* The filter slides across every position in the image
* Same filter weights are used everywhere (**parameter sharing**)

Example: A 3x3 edge-detection filter might have weights [[-1,0,1],[-2,0,2],[-1,0,1]]. When convolved with an image, it produces high values at vertical edges.

3.3 Key CNN Components

Component	Purpose	Effect
Convolution Layer	Detect local patterns	Creates feature maps
ReLU Activation	Add nonlinearity	Enables complex patterns
Pooling Layer	Downsample, add invariance	Reduces size, robust to shifts
Fully Connected	Final classification	Combines all features

3.4 Pooling Layers

Max Pooling: Take the maximum value in each small region (e.g., 2x2 block). This:

* Reduces spatial dimensions (2x2 pooling halves width and height)
* Provides **translation invariance** - small shifts don't change the max
* Reduces computation for subsequent layers

3.5 Typical CNN Architecture

A typical CNN repeats blocks of: **Conv -> ReLU -> Pool**

Input (224x224x3) -> [Conv->ReLU->Pool] -> [Conv->ReLU->Pool] -> [Conv->ReLU->Pool] -> Flatten -> Dense -> Softmax -> Output (1000 classes)

Feature hierarchy: Early layers have small receptive fields and detect simple patterns. Later layers have larger receptive fields (through pooling) and detect complex patterns composed of simpler ones.

3.6 Data Augmentation

To prevent overfitting, artificially expand the training set by applying random transformations:

* Random rotations (up to 15 degrees)
* Random shifts (left/right, up/down)
* Random zooming and cropping
* Horizontal flipping
* Color jittering

Intuition: A slightly rotated cat is still a cat. Augmentation teaches the network this invariance without needing more labeled data.

4. Recurrent Neural Networks (RNNs)

4.1 The Problem with Sequential Data

Standard neural networks assume inputs are independent. But for sequences (text, speech, time series), **order matters** and **context is crucial**.

Example: In the sentence 'The bank by the river', the meaning of 'bank' depends on the word 'river' that comes later. We need a network that can remember and use context.

4.2 The RNN Architecture

RNNs process sequences one element at a time, maintaining a **hidden state** that carries information from previous steps:

$$A_t = g(WA_{t-1} + UX_t + b)$$

Where:

* A_t = hidden state at time t (the 'memory')
* X_t = input at time t
* W = weights for previous hidden state (recurrent connection)
* U = weights for current input
* The **same weights** W, U are used at every time step

Intuition: At each step, the RNN combines 'what I remember' (W*A_{t-1}) with 'what I see now' (U*X_t) to form a new memory state.

4.3 Applications of RNNs

Application	Input Sequence	Output
Sentiment Analysis	Words in a review	Positive/Negative
Language Modeling	Words so far	Next word
Machine Translation	Source sentence	Target sentence
Speech Recognition	Audio frames	Text
Time Series	Past values	Future prediction

4.4 Word Embeddings

Words are typically represented as **embeddings** - dense vectors of maybe 100-300 dimensions that capture semantic meaning:

* Similar words have similar embeddings (cat and dog are close)
* Captures relationships: king - man + woman ≈ queen
* Much more efficient than one-hot encoding (vocab size can be 50,000+)

4.5 The Vanishing Gradient Problem

For long sequences, gradients can become very small (vanish) or very large (explode) when backpropagating through many time steps. Early inputs have little influence on later outputs.

Solution: LSTM (Long Short-Term Memory):

LSTMs add 'gates' that control what information to keep, forget, or output. This allows gradients to flow more easily through long sequences.

5. Fitting Neural Networks

5.1 The Non-Convexity Problem

Unlike linear regression or logistic regression, the loss function for neural networks is **non-convex** - it has many local minima. The solution we find depends on:

* Random weight initialization
* Learning rate and optimizer choice
* Order of training examples

Good news: In practice, different local minima often have similar test performance. Finding THE global minimum isn't necessary.

5.2 Gradient Descent

We minimize the loss by iteratively moving in the direction of steepest descent:

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla L(\theta^{(t)})$$

where eta is the **learning rate** - how big a step to take.

5.3 Backpropagation

How do we compute gradients for millions of parameters? **Backpropagation** efficiently applies the chain rule:

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial A} \cdot \frac{\partial A}{\partial w}$$

Starting from the output, we compute gradients layer by layer, reusing intermediate computations. This is just calculus chain rule applied systematically!

5.4 Stochastic Gradient Descent (SGD)

Computing gradients on the full dataset is expensive. Instead, use a random **minibatch** (e.g., 32-256 examples) at each step:

* Much faster per update
* Noisy gradients can help escape local minima
* An **epoch** = one full pass through all training data

Common optimizers: SGD with momentum, Adam, RMSprop. These adapt the learning rate for each parameter based on past gradients.

6. Interpolation and Double Descent

6.1 The Classical View

Traditional bias-variance trade-off says: As model complexity increases, training error decreases but test error follows a U-shape (first decreases, then increases due to overfitting).

6.2 The Surprising Discovery

Deep learning often violates this! Test error can exhibit **double descent**:

1. Initially, test error decreases with complexity (classical regime)
2. At 'interpolation threshold' (model just fits training data perfectly), test error peaks
3. Beyond this, test error **decreases again** as model becomes more over-parameterized!

Key insight: Highly over-parameterized models trained with SGD tend to find 'smooth' solutions among the many that perfectly fit the training data. These smooth solutions generalize well.

7. When to Use Deep Learning

Deep learning is not always the answer. Consider the trade-offs:

Use Deep Learning When...	Use Simpler Methods When...
Large dataset (millions of examples)	Small dataset (hundreds/thousands)
Complex structure (images, text, audio)	Tabular data with clear features
Interpretability not critical	Need to explain predictions
Compute resources available	Limited compute budget
State-of-the-art accuracy needed	Good-enough accuracy is fine

Occam's Razor: For tabular data (like the Hitters salary prediction), linear models, Lasso, or random forests often perform just as well as neural networks but are much easier to fit, tune, and interpret. Choose the simplest model that does the job!

8. Key Takeaways

* **Neural networks** learn hierarchical representations through layers of nonlinear transformations.
* **Activation functions** (ReLU, sigmoid, tanh) provide essential nonlinearity.
* **Softmax + cross-entropy** for classification; squared error for regression.
* **Regularization is essential**: dropout, weight decay, early stopping, data augmentation.
* **CNNs** exploit spatial structure with convolutions and pooling - ideal for images.
* **RNNs/LSTMs** handle sequential data by maintaining hidden state across time steps.
* **Backpropagation** efficiently computes gradients via chain rule.
* **SGD** with minibatches makes training large networks feasible.
* **Double descent**: over-parameterized models can generalize well despite interpolating training data.
* Deep learning excels on large, complex datasets; use simpler methods for tabular/small data.

Quick Reference: Key Formulas

Hidden unit activation:

$$A_k = g\left(w_{k0} + \sum_{j=1}^{p} w_{kj} X_j\right)$$

ReLU:

$$g(z) = (z)_+ = \max(0, z)$$

Softmax:

$$P(Y=k|X)=\frac{e^{Z_k}}{\sum_{l=1}^{K} e^{Z_l}}$$

RNN hidden state:

$$A_t = g(WA_{t-1} + UX_t + b)$$