

- **Course:** CSCI E-89B: Introduction to Natural Language Processing
- **Lecture:** Lecture 01 – Neural Networks Foundations
- **Instructor:** Dmitry Kurochkin
- **Objective:** Understand the fundamentals of neural networks including architecture, activation functions, loss functions, cost functions, and optimization algorithms for NLP applications

Contents

1 Introduction: Course Overview and Learning Roadmap

Lecture Overview

Welcome to CSCI E-89B: Introduction to Natural Language Processing. This course covers the intersection of deep learning and language processing. Before diving into NLP-specific techniques, we must first establish a strong foundation in neural networks.

Key Learning Objectives:

- Understand the fundamental architecture of neural networks
- Learn about activation functions and their role in introducing non-linearity
- Distinguish between loss functions (individual error) and cost functions (aggregate error)
- Master gradient descent and its variants for optimization
- Apply these concepts using Keras/TensorFlow

1.1 Why Neural Networks for NLP?

Key Information

The Feature Engineering Problem

Traditional machine learning requires **manual feature engineering**—humans must design the features that the model uses.

Example: Polynomial Regression

$$\hat{y} = w_0 + w_1 \cdot x + w_2 \cdot x^2 + w_3 \cdot x^3 \quad (1)$$

Here, x , x^2 , and x^3 are features we *manually* created. This works for simple problems, but:

- Images have millions of pixels with complex relationships
- Text has sequences of words with semantic meaning
- How do you manually design features for “the meaning of a sentence”?

Neural networks solve this by learning features automatically from data.

1.2 Course Communication Channels

Channel	Purpose	Notes
Piazza	Official Q&A forum	Instructors monitor and respond; share with class
WhatsApp	Informal student discussions	Not officially monitored; for peer support
Canvas Inbox	Private communication	For personal matters with instructors/TAs

Table 1: Course Communication Channels

1.3 Weekly Schedule

Session	Day	Focus
Lecture	Tuesday	Theory and core concepts
TA Session 1	Wed/Thursday	Theory review, problem solving
Instructor's Section	Friday	Python implementations, code examples
TA Session 2	Sat/Sunday	Additional problems (different content from Session 1)

Table 2: Weekly Session Schedule

Warning

Important Policy Notes:

- **Quizzes:** No late submissions allowed. Solutions are discussed immediately after the deadline.
- **Assignments:** Due Sundays 11:59 PM Boston time. Late penalty: 10% per day.
- **Final Project:** Strict deadline—extensions require official Extension School paperwork.
- **No dropping:** Unlike some courses, the lowest quiz/assignment is NOT dropped.

1.4 Grading Breakdown

Component	Weight
Weekly Assignments	65%
Weekly Quizzes	20%
Final Project	15%

Table 3: Grade Distribution

2 From Linear Regression to Neural Networks

2.1 The Limitations of Linear Models

Definition:

Linear Regression **Linear regression** fits a model that is *linear in its parameters*:

$$\hat{y} = w_0 + w_1 \cdot u_1 + w_2 \cdot u_2 + w_3 \cdot u_3 \quad (2)$$

where u_1, u_2, u_3 can be transformed versions of the input (e.g., $u_1 = x, u_2 = x^2, u_3 = x^3$).

The Problem:

- Features (u_1, u_2, u_3) must be **manually designed**
- Works for simple data where you can visualize and understand relationships
- Fails for high-dimensional data (images, text, audio)

Why not make the powers learnable?

You might think: “Let’s learn the optimal power p in x^p ”

$$\hat{y} = w_0 + w_1 \cdot x^{p_1} + w_2 \cdot x^{p_2} + w_3 \cdot x^{p_3} \quad (3)$$

This is a *terrible idea* because:

- The function becomes highly non-linear in parameters
- Optimization becomes extremely difficult (many bad local minima)
- Derivatives with respect to p are complex

2.2 The Neural Network Solution

Important:

The Key Insight Neural networks introduce non-linearity through **activation functions** applied to **linear combinations**.

$$u_1 = f(w_0 + w_1 x_1 + w_2 x_2) \quad (4)$$

Why this specific form?

- Linear combinations are easy to differentiate
- Chain rule applies cleanly
- We can use gradient descent efficiently

The derivative:

$$\frac{\partial u_1}{\partial w} = f'(\cdot) \cdot \frac{\partial}{\partial w} (w_0 + w_1 x_1 + w_2 x_2) \quad (5)$$

The derivative of the linear part is trivial, and f' is usually simple too.

3 Feedforward Neural Networks

Definition:

Feedforward Neural Network (FNN) A **feedforward neural network** is a composition of functions where information flows in one direction—from input to output. Mathematically:

$$\hat{y} = f^{(L)} \left(f^{(L-1)} \left(\dots f^{(1)}(x) \right) \right) \quad (6)$$

Each function $f^{(l)}$ typically consists of a linear transformation followed by a non-linear activation.

3.1 A Simple Two-Layer Network

Consider a network with:

- 2 inputs: x_1, x_2
- 2 hidden neurons: u_1, u_2
- 1 output: \hat{y}

Hidden Layer Computation:

$$u_1 = f \left(w_{01}^{(1)} + w_{11}^{(1)} x_1 + w_{21}^{(1)} x_2 \right) \quad (7)$$

$$u_2 = f \left(w_{02}^{(1)} + w_{12}^{(1)} x_1 + w_{22}^{(1)} x_2 \right) \quad (8)$$

Output Layer Computation:

$$\hat{y} = f \left(w_0^{(2)} + w_1^{(2)} u_1 + w_2^{(2)} u_2 \right) \quad (9)$$

Key Information

Understanding the Notation:

- $w_{ij}^{(l)}$: Weight connecting input i to neuron j in layer l
- $w_{0j}^{(l)}$: Bias term for neuron j in layer l
- f : Activation function

3.2 The Role of the Bias Term

Example:

Why Do We Need Bias? The bias term w_0 acts as a **threshold shifter**.

Consider a biological analogy: A neuron “fires” when the cumulative input exceeds a threshold. Without bias, this threshold is fixed at zero. With bias, we can adjust where the activation “turns on.”

Mathematically, instead of:

$$f(w_1 x_1 + w_2 x_2) \quad (\text{threshold at } 0) \quad (10)$$

We have:

$$f(w_0 + w_1x_1 + w_2x_2) \quad (\text{adjustable threshold}) \quad (11)$$

The bias allows the decision boundary to shift away from the origin.

3.3 Universal Approximation Theorem

Theorem 3.1 (Universal Approximation). A feedforward neural network with a single hidden layer containing a finite number of neurons can approximate any continuous function on a compact subset of \mathbb{R}^n , given an appropriate activation function.

Implications:

- Theoretically, one layer is “enough” for any function
- In practice, deeper networks learn more efficiently
- The choice of activation function matters less theoretically, but significantly in practice

4 Activation Functions

Definition:

Activation Function An **activation function** is a non-linear function applied to the output of a neuron. Without activation functions, stacking linear layers would result in just another linear transformation—unable to learn complex patterns.

4.1 Why Non-linearity is Essential

If we had no activation function:

$$u = W^{(1)}x + b^{(1)} \quad (12)$$

$$\hat{y} = W^{(2)}u + b^{(2)} = W^{(2)}(W^{(1)}x + b^{(1)}) + b^{(2)} \quad (13)$$

This collapses to:

$$\hat{y} = \underbrace{W^{(2)}W^{(1)}}_{W'} x + \underbrace{W^{(2)}b^{(1)} + b^{(2)}}_{b'} \quad (14)$$

A single linear transformation! No matter how many layers, without non-linearity, we cannot learn complex patterns.

4.2 Biological Inspiration: The Step Function

Warning

Historical Note: The Step Function

Early neural networks mimicked biological neurons using the **Heaviside step function**:

$$f(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases} \quad (15)$$

Problem: The derivative is zero everywhere (except at $z = 0$ where it's undefined).

This means the cost function becomes **piecewise constant**—gradient descent doesn't know which direction to move! This activation is NOT used in practice.

4.3 Common Activation Functions

4.3.1 ReLU (Rectified Linear Unit)

$$f(z) = \max(0, z) = \begin{cases} z & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases} \quad (16)$$

Advantages:

- Computationally efficient

Function	Formula	Range	Use Case
ReLU	$f(z) = \max(0, z)$	$[0, \infty)$	Hidden layers (most common)
Leaky ReLU	$f(z) = \max(\alpha z, z)$	$(-\infty, \infty)$	Hidden layers (avoids dead neurons)
Sigmoid	$f(z) = \frac{1}{1+e^{-z}}$	$(0, 1)$	Binary classification output
Softmax	$f(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$	$(0, 1)$, sum = 1	Multi-class classification output
Tanh	$f(z) = \tanh(z)$	$(-1, 1)$	Hidden layers, RNNs

Table 4: Common Activation Functions

- Doesn't saturate for positive values
- Widely used and well-studied

Disadvantage:

- “Dead neurons”: If $z < 0$, gradient is 0, neuron stops learning

4.3.2 Leaky ReLU

$$f(z) = \max(\alpha z, z) \quad \text{where } \alpha \approx 0.01 - 0.1 \quad (17)$$

The small slope for negative values prevents dead neurons. However, α is a **hyperparameter** that must be chosen.

4.3.3 Softmax for Classification**Definition:**

Softmax Function For a vector $\mathbf{z} = [z_1, z_2, \dots, z_M]$:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^M e^{z_j}} \quad (18)$$

Properties:

- Each output is in $(0, 1)$
- All outputs sum to exactly 1
- Outputs can be interpreted as probabilities

Example:

Softmax Calculation Given logits $\mathbf{z} = [11, 10]$:

$$\text{softmax}(z_1) = \frac{e^{11}}{e^{11} + e^{10}} = \frac{e^{11}}{e^{11}(1 + e^{-1})} \approx 0.73 \quad (19)$$

$$\text{softmax}(z_2) = \frac{e^{10}}{e^{11} + e^{10}} \approx 0.27 \quad (20)$$

The larger input gets the higher probability. The “winning” class is amplified.

4.4 Keras Implementation

```

1 import keras
2 from keras import models, layers
3
4 model = models.Sequential()
5
6 # Hidden layer: 16 neurons with ReLU activation
7 # Input shape: 900 features
8 model.add(layers.Dense(16, activation='relu', input_shape=(900,)))
9
10 # Output layer: 2 neurons with Softmax (for binary classification)
11 model.add(layers.Dense(2, activation='softmax'))
12
13 model.summary()
```

Listing 1: Building a Neural Network in Keras

5 Loss Functions and Cost Functions

Definition:

Loss vs Cost **Loss Function** $L^{(i)}(w)$: Measures the error for a **single** data point $(x^{(i)}, y^{(i)})$.

Cost Function $J(w)$: The **average** loss over the entire dataset:

$$J(w) = \frac{1}{m} \sum_{i=1}^m L^{(i)}(w) \quad (21)$$

Analogy:

- Loss = How rotten is *one* apple?
- Cost = On average, how rotten is the entire box of apples?

5.1 Loss Functions for Regression

When predicting continuous values:

5.1.1 Squared Error (SE)

$$L(\hat{y}, y) = (\hat{y} - y)^2 \quad (22)$$

Properties:

- Penalizes large errors more heavily
- Differentiable everywhere
- Sensitive to outliers

When averaged: **Mean Squared Error (MSE)**:

$$J(w) = \frac{1}{m} \sum_{i=1}^m (\hat{y}^{(i)} - y^{(i)})^2 \quad (23)$$

5.1.2 Absolute Error (AE)

$$L(\hat{y}, y) = |\hat{y} - y| \quad (24)$$

Properties:

- Less sensitive to outliers
- Not differentiable at $\hat{y} = y$
- Constant gradient magnitude

When averaged: **Mean Absolute Error (MAE)**

Key Information

MSE vs MAE: When to Use Which?

Both have the same minimum (the correct prediction), but their **gradients behave differently**:

- **MSE:** Gradient gets smaller as you approach the minimum (self-adjusting step sizes)
- **MAE:** Constant gradient (fixed step size regardless of distance to minimum)

The shape of the cost function affects which local minimum you converge to!

5.2 Loss Functions for Classification

5.2.1 Why Not Use MSE for Classification?

Warning

MSE is Terrible for Classification!

Consider classifying images as Cat (1,0) or Dog (0,1):

- True label: $y = [1, 0]$ (Cat)
- Prediction: $\hat{y} = [0.9, 0.1]$

MSE computes: $(0.9 - 1)^2 + (0.1 - 0)^2 = 0.01 + 0.01 = 0.02$

The Problem:

- y lives in a discrete space: $\{[1, 0], [0, 1]\}$
- \hat{y} lives on a continuous line (probabilities summing to 1)
- Computing distance between discrete and continuous spaces creates a highly non-convex cost function
- Result: You get stuck in bad local minima and fail to converge

5.2.2 Cross-Entropy Loss

Definition:

Cross-Entropy For a classification problem with M classes:

$$L(\hat{y}, y) = - \sum_{j=1}^M y_j \log(\hat{y}_j) \quad (25)$$

For binary classification (with one-hot encoding):

$$L = -y_1 \log(\hat{y}_1) - y_2 \log(\hat{y}_2) \quad (26)$$

Since y is one-hot encoded (e.g., $[1, 0]$), only one term contributes.

Example:

Cross-Entropy Calculation Case 1: Good Prediction

- True: $y = [1, 0]$ (Cat)
- Predicted: $\hat{y} = [0.99, 0.01]$

- Loss: $-1 \cdot \log(0.99) - 0 \cdot \log(0.01) \approx 0.01$

Case 2: Bad Prediction

- True: $y = [1, 0]$ (Cat)
- Predicted: $\hat{y} = [0.01, 0.99]$
- Loss: $-1 \cdot \log(0.01) - 0 \cdot \log(0.99) \approx 4.6$

Cross-entropy heavily penalizes confident wrong predictions!

Key Information

Why Cross-Entropy Works:

- When prediction matches truth: $\log(1) = 0$ (zero loss)
- When prediction is wrong: $\log(\epsilon)$ becomes very negative (high loss)
- The cost function has a much nicer shape for optimization
- Softmax ensures predictions are never exactly 0, avoiding $\log(0)$

5.3 Keras Implementation

```

1 model.compile(
2     optimizer='adam',                      # Optimization algorithm
3     loss='categorical_crossentropy',        # Cross-entropy for multi-class
4     metrics=['accuracy']                  # What to display during training
5 )
6
7 # For binary classification with sigmoid output:
8 # loss='binary_crossentropy'
9
10 # For regression:
11 # loss='mse' or loss='mae'
```

Listing 2: Specifying Loss and Optimizer in Keras

6 Optimization: Gradient Descent

Definition:

Gradient Descent **Gradient descent** is an iterative algorithm to find the minimum of a function by repeatedly moving in the direction of steepest descent (negative gradient).

Update Rule:

$$w_{\text{new}} = w_{\text{old}} - \alpha \nabla J(w_{\text{old}}) \quad (27)$$

Where:

- α : Learning rate (step size)
- ∇J : Gradient of the cost function

6.1 The Gradient

Definition:

Gradient The **gradient** ∇J is a vector of partial derivatives:

$$\nabla J = \left[\frac{\partial J}{\partial w_1}, \frac{\partial J}{\partial w_2}, \dots, \frac{\partial J}{\partial w_d} \right] \quad (28)$$

It points in the direction of **steepest increase** of J . We move in the **opposite** direction to minimize.

6.2 Variants of Gradient Descent

Variant	Batch Size	Pros	Cons
Gradient Descent (GD)	All m samples	Stable, true gradient	Very slow for large datasets
Stochastic GD (SGD)	1 sample	Fast updates, escapes local minima	Very noisy, slow convergence
Mini-batch GD	n samples (e.g., 32)	Best of both worlds	Requires tuning batch size

Table 5: Gradient Descent Variants

6.2.1 Batch Gradient Descent

$$\nabla J = \frac{1}{m} \sum_{i=1}^m \nabla L^{(i)} \quad (29)$$

Issues:

- Must compute gradient over ALL data points before one update
- Very expensive for large datasets
- Deterministic—no randomness to escape local minima

6.2.2 Stochastic Gradient Descent (SGD)

$$\nabla J \approx \nabla L^{(i)} \quad (\text{single random sample}) \quad (30)$$

Properties:

- Very fast per update
- Highly stochastic—helps escape local minima
- Too noisy—bounces around a lot
- Cannot be easily parallelized (each step depends on previous)

6.2.3 Mini-batch Gradient Descent

$$\nabla J \approx \frac{1}{n} \sum_{i \in \text{batch}} \nabla L^{(i)} \quad (31)$$

The Sweet Spot:

- Typical batch sizes: 32, 64, 128, 256
- Still stochastic (can escape local minima)
- Parallelizable (all samples in batch use same weights)
- The “soup and spoon” analogy: You don’t need a bigger spoon for a bigger pot

Key Information

The Soup and Spoon Analogy

To check if soup is salty enough:

- You don’t need to drink the entire pot
- A small spoon gives you a good estimate
- The spoon size doesn’t need to scale with pot size

Similarly, a mini-batch of 32-64 samples is enough to estimate the gradient, regardless of whether your dataset has 10,000 or 10 million samples!

6.3 The Non-Convex Optimization Challenge

Important:

Why Deep Learning Optimization is Hard Neural network cost functions are **highly non-convex**:

- Many local minima (not just one global minimum)
- Saddle points (flat regions with zero gradient)
- The landscape depends on architecture, data, and loss function

Implications:

- Different random initializations → different solutions
- Different students solving the same problem may get different results
- This is why we share student solutions—to see what different approaches find!

6.4 Learning Rate Considerations

Warning

Choosing the Learning Rate α :

- **Too large:** Algorithm diverges (overshoots the minimum)
- **Too small:** Convergence is extremely slow
- **Just right:** Fast convergence to a good minimum

MSE has a nice property: As you approach the minimum, the gradient gets smaller, so steps automatically become smaller. This is “self-adjusting.”

MAE is trickier: The gradient magnitude is constant, so step sizes don’t decrease near the minimum.

6.5 Advanced Optimizers

Optimizer	Description
SGD	Basic stochastic gradient descent
SGD + Momentum	Accumulates past gradients for smoother updates
Adagrad	Adapts learning rate per parameter based on history
RMSprop	Improves Adagrad by using moving average
Adam	Combines momentum + adaptive learning rates; default choice

Table 6: Common Optimizers

```

1 from keras.optimizers import SGD, Adam
2
3 # Basic SGD
4 model.compile(optimizer=SGD(learning_rate=0.01), loss='mse')
5
6 # SGD with momentum
7 model.compile(optimizer=SGD(learning_rate=0.01, momentum=0.9), loss='mse')
8
9 # Adam (recommended default)
10 model.compile(optimizer=Adam(learning_rate=0.001), loss=
11     'categorical_crossentropy')
12
13 # Training with mini-batch
14 history = model.fit(
15     X_train, y_train,
16     batch_size=64,           # Mini-batch size
17     epochs=50,              # Number of full passes through data
18     validation_data=(X_val, y_val)
19 )

```

Listing 3: Using Different Optimizers in Keras

7 Putting It All Together: Training a Neural Network

7.1 The Complete Training Pipeline

1. **Initialize weights** randomly
2. **Forward pass:** Compute predictions \hat{y}
3. **Compute loss:** Measure error using loss function
4. **Backward pass:** Compute gradients via backpropagation
5. **Update weights:** Apply optimizer (e.g., Adam)
6. **Repeat** until convergence or max epochs

7.2 Complete Keras Example

```

1 import keras
2 from keras import models, layers
3 from keras.optimizers import Adam
4
5 # 1. Build the model
6 model = models.Sequential([
7     layers.Dense(16, activation='relu', input_shape=(900,)),    # Hidden
8         layer
9     layers.Dense(2, activation='softmax')                         # Output
10    layer
11])
12
13 # 2. Compile with loss, optimizer, and metrics
14 model.compile(
15     optimizer=Adam(learning_rate=0.001),
16     loss='categorical_crossentropy',
17     metrics=['accuracy']
18 )
19
20 # 3. Train the model
21 history = model.fit(
22     X_train, y_train,
23     batch_size=128,
24     epochs=35,
25     validation_data=(X_test, y_test)
26 )
27
28 # 4. Evaluate
29 test_loss, test_accuracy = model.evaluate(X_test, y_test)
30 print(f"Test Accuracy: {test_accuracy:.4f}")

```

Listing 4: Complete Neural Network Training Example

7.3 Understanding the Output

- **Training Loss:** Should decrease over epochs
- **Validation Loss:** Should also decrease; if it increases, you're overfitting
- **Accuracy:** Metric for monitoring, not for optimization

Glossary

Term	Definition
Neural Network	A model inspired by biological neurons that learns features from data
Activation Function	Non-linear function applied to neuron outputs (e.g., ReLU, Softmax)
Loss Function	Measures error for a single data point
Cost Function	Average loss over the entire dataset
Gradient Descent	Optimization algorithm that follows the negative gradient
Learning Rate	Step size for weight updates (α)
Mini-batch	Subset of data used for each gradient update
Epoch	One complete pass through the training data
One-Hot Encoding	Representing categories as binary vectors (e.g., Cat \rightarrow [1,0])
Hyperparameter	Parameters set before training (e.g., learning rate, batch size)
Cross-Entropy	Loss function for classification problems
Softmax	Activation that converts logits to probabilities

One-Page Summary

CSCI E-89B Lecture 01: Neural Networks Foundations

1. Why Neural Networks?

- Manual feature engineering doesn't scale to complex data (images, text)
- NNs learn features automatically through nested non-linear transformations

2. Architecture

- Layers of neurons: Input → Hidden → Output
- Each neuron: $u = f(w_0 + w_1x_1 + w_2x_2 + \dots)$
- Linear combination + non-linear activation

3. Activation Functions

- Hidden layers: ReLU $\max(0, z)$ or Leaky ReLU
- Binary classification output: Sigmoid $\frac{1}{1+e^{-z}}$
- Multi-class output: Softmax $\frac{e^{z_i}}{\sum e^{z_j}}$

4. Loss Functions

- Regression: MSE $(y - \hat{y})^2$ or MAE $|y - \hat{y}|$
- Classification: Cross-Entropy $-\sum y_j \log(\hat{y}_j)$
- Never use MSE for classification!

5. Optimization

- Gradient Descent: $w_{\text{new}} = w_{\text{old}} - \alpha \nabla J$
- Mini-batch GD: Best balance of speed and stability (batch size 32-64)
- Default optimizer: Adam

6. Key Formulas

$$\text{Cost Function: } J(w) = \frac{1}{m} \sum_{i=1}^m L^{(i)}(w)$$

$$\text{Cross-Entropy: } L = - \sum_{j=1}^M y_j \log(\hat{y}_j)$$

$$\text{Softmax: } \hat{y}_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$$