2025_ML_assignment3_written

Kai-Hung Cheng

2025/09/23

1 Introduction

In calculus, we learn that polynomials can be used to approximate other functions. For example, the Taylor expansion lets us write a smooth function as a polynomial near a point. A more general result, the Weierstrass Approximation Theorem, says that *any continuous function* on a closed interval can be approximated by a polynomial as closely as we want.

Neural networks turn out to be another way of approximating functions. A neural network works by taking an input, multiplying by weights, adding biases, and then applying a nonlinear function called an **activation function**. By stacking many of these simple steps together, neural networks can create very complicated shapes.

In this report, we look at two key results (Lemma 3.1 and Lemma 3.2) from the paper by Ryck et al., *On the approximation of functions by tanh neural networks*. These lemmas show that even very simple neural networks (with only one hidden layer) using the tanh function as activation can already approximate polynomials very well. Since polynomials are the "building blocks" for approximating general functions, this helps explain why neural networks are so powerful.

2 Background Concepts

Before diving into the lemmas, let's review a few basic ideas.

(a) The tanh function

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

This function looks like a smooth S-shaped curve, starting near -1 when x is very negative and going up to 1 when x is very positive. It is also an **odd function**, meaning $\tanh(-x) = -\tanh(x)$.

(b) Structure of a neural network

A shallow neural network (only one hidden layer) works like this:

1. Take the input and apply a linear transformation (multiply by a weight and add a bias).

2. Apply the tanh function to bend the line into a curve.
3. Add together many such curves to create a more complicated output.

Think of it like using Lego blocks: each tanh curve is a small curved block, and by combining them, we can build different shapes.

(c) Error measurement (Sobolev norm)

When we measure how well one function approximates another, we might look at the difference between their values. But sometimes we also care about the slopes and higher derivatives. A Sobolev norm is just a way of measuring the error that takes both the function and its derivatives into account. Intuitively, it means the approximation not only follows the shape of the curve, but also matches its tilt and curvature.

3 Lemma 3.1 (Odd Degree Polynomials)

What the lemma says.

If s is an odd number, then we can build a shallow tanh network with about $\frac{s+1}{2}$ neurons that can approximate all odd power functions y^p for $p \leq s$. For example, y^3, y^5, y^7, etc., all the way up to s.

Why this makes sense.

The tanh function itself is an odd curve. That makes it naturally suited for building other odd-shaped curves. By shifting and scaling $\tanh(x)$, then adding them together, we can bend the network output so it looks like y^3, y^5, and so on. Adding more neurons means we have more "pieces" to work with, so we can make the approximation as close as we want.

Why it matters.

This lemma shows that even very simple networks can already handle a wide family of power functions (the odd powers). Since power functions are the building blocks of polynomials, this is the first step toward approximating more general functions.

4 Lemma 3.2 (All Polynomials up to s)

What the lemma says.

Lemma 3.2 goes one step further. It shows that with about $\frac{3(s+1)}{2}$ neurons, a shallow tanh network can approximate *all* power functions up to degree s, including both odd and even powers. That means it can handle $y, y^2, y^3, \ldots, y^s$.

Why this makes sense.

We already know from Lemma 3.1 how to build odd powers. For even powers like y^2 or y^4, the authors use clever formulas that express them in terms of odd powers plus lower-degree terms. Since the network can already handle the odd cases, we can "bootstrap" our way to the even ones. It just takes a bit more width (roughly 1.5 times more neurons) to cover all cases.

Why it matters.

Now we have everything: the network can approximate any polynomial up to degree s. And thanks to the Weierstrass Approximation Theorem, we know polynomials can approximate any continuous function. Put together:

- Lemma 3.1: handles odd power functions
- Lemma 3.2: extends to all power functions
- Therefore: shallow tanh networks can approximate any continuous function

This is why people say neural networks are **universal approximators**.

5 Conclusion

We looked at two lemmas that explain how shallow tanh networks can approximate functions. Lemma 3.1 shows they can build odd power functions like $y^3, y^5, \ldots$. Lemma 3.2 shows that, with slightly more neurons, they can also build even power functions, and therefore any polynomial up to degree s.

Since polynomials can approximate any continuous function, these lemmas tell us that shallow tanh networks are universal approximators. The big picture is simple: by combining a handful of S-shaped tanh curves, we can mimic the shape of polynomials as closely as we want. This gives both an intuitive and a mathematical reason why neural networks are so effective in practice.