Why Bigger MLPs Aren't Always Better: A Practical Study on Depth, Width, Overfitting, and Gradient Issues

1 Introduction to Multi-Layer Perceptrons (MLPs)

A *Multi-Layer Perceptron* (MLP)—also known as a feedforward neural network or a simple deep neural network—is one of the fundamental architectures used in modern deep learning. An MLP consists of a sequence of layers through which data moves strictly forward, from the input to the output.

1.1 Network Architecture

An MLP is built from at least three layers:

- **Input Layer:** Receives the feature vector $\mathbf{x}$ and serves as the entry point to the network.
- **Hidden Layer(s):** One or more intermediate layers that transform the input using nonlinear operations. For a hidden layer k, the computation is

$$\mathbf{h}^{(k)} = \sigma\Big(W^{(k)}\mathbf{h}^{(k-1)} + \mathbf{b}^{(k)}\Big),$$

  where $W^{(k)}$ and $\mathbf{b}^{(k)}$ denote the weights and biases, and $\sigma(\cdot)$ is a nonlinear activation function such as ReLU or sigmoid.
- **Output Layer:** Produces the final prediction, such as class probabilities or a continuous output for regression. It is typically written as

$$\mathbf{y} = f\Big(W^{(L)}\mathbf{h}^{(L-1)} + \mathbf{b}^{(L)}\Big),$$

  with $f(\cdot)$ chosen based on the task (e.g., softmax for classification).

1.2 Fully Connected Structure

Each neuron in one layer connects to every neuron in the next, forming a *fully connected* (dense) architecture. Each connection carries a trainable weight, and each neuron includes a bias term.

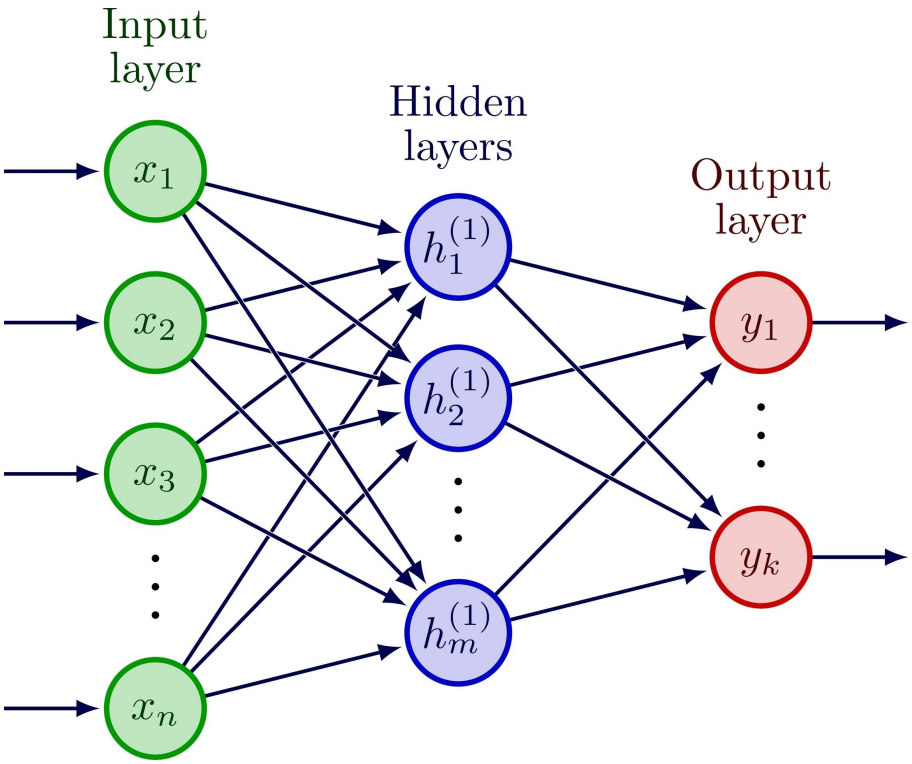

Figure 1: Illustration of a basic MLP structure.

1.3 How an MLP Processes Information

During the forward pass, every layer performs the following operations:

1. Compute a weighted sum of the outputs from the previous layer.
2. Add a bias term.
3. Apply a nonlinear activation function:

$$z = W\mathbf{x} + \mathbf{b}, \qquad a = \sigma(z).$$

The inclusion of nonlinear activation functions enables the network to learn complex relationships that cannot be captured by linear models.

1.4 Training the Network

MLPs learn by adjusting weights and biases to minimize a chosen loss function. Training typically relies on:

- **Backpropagation** to compute gradients, and
- **Optimization algorithms** such as Stochastic Gradient Descent (SGD) or Adam to update the parameters.

Through repeated iterations, the model gradually improves its ability to generalize from training data to unseen inputs.

2 Dataset

The digit recognizer dataset is provided as a structured CSV file in which each record contains a numeric class label (0–9) followed by a 784-dimensional feature vector representing the pixel intensities of a handwritten digit. Each value corresponds to a grayscale pixel from a 28×28 image that has been flattened into a single row, with intensity levels ranging from 0 to 255. The dataset contains 42,000 training samples, which are divided into training, validation, and test sets with a 70/20/10 ratio. Pixel intensities are normalized to [0, 1] to improve neural network training. This high-dimensional, structured representation makes the dataset ideal for experimenting with multi-layer perceptrons and for analyzing how variations in network depth and width affect learning performance.

3 MLP Architecture

Below is the function used to create a fully connected network (MLP), where both depth and width can be dynamically adjusted, shown in Figure 2:

- **Depth controls the number of hidden layers:** The depth parameter determines how many fully connected layers are stacked. Increasing depth makes the network "deeper," which can increase representational power but may also make training harder (e.g., vanishing gradients).
- **Width controls the number of neurons per layer:** The width parameter defines how many neurons each hidden layer contains. Wider networks can capture more features per layer but can overfit more easily.
- **Using ReLU activation:** Each hidden layer uses the ReLU function, which mitigates vanishing gradients compared to sigmoids but very deep MLPs can still experience them.
- **Softmax output layer:** The final layer uses softmax activation to produce probabilities for num_classes categories, suitable for classification tasks.

```python
def build_mlp(depth=3, width=256, num_classes=10):
    model = models.Sequential()
    # 1) Input layer
    model.add(layers.Input(shape=(784,)))
    # 2) Hidden layers
    for _ in range(depth):
        model.add(layers.Dense(width, activation="relu"))
    # 3) Output layer
    model.add(layers.Dense(num_classes, activation="softmax"))
    # 4) Compile
    model.compile(
        optimizer="adam",
        loss="sparse_categorical_crossentropy",
        metrics=["accuracy"]
    )
    return model
```

Figure 2: Dynamic MLP function allowing flexible depth and width. Hidden layers use ReLU activation, and the output layer uses softmax for multi-class classification.

4 Experiment Setup

We design experiments to study the effect of depth and width on training dynamics and performance of MLPs. Both experiments use the `train_mlp` function shown in Figure 3. For the depth experiment, we vary the number of hidden layers (e.g., [2, 4, 8, 16, 32]), keeping the neurons per layer fixed (e.g., 128). For the width experiment, we vary the number of neurons per hidden layer (e.g., [32, 64, 128, 256, 512, 1024]), keeping the number of layers fixed (e.g., 3). We track training time per epoch, final accuracy, and total parameters for both experiments to understand the trade-offs. **Hyperparameters Used:**

- **Optimizer:** Adam
- **Learning Rate:** 0.001 (Adam default)
- **Adam Parameters:** $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-7}$
- **Loss Function:** Sparse categorical cross-entropy
- **Metrics:** Accuracy
- **Epochs:** 10
- **Batch Size:** 32 (Keras default)
- **Weight Initialization:** Glorot Uniform (Keras default)
- **Activation Functions:** ReLU for hidden layers, Softmax for output layer

```python
def train_mlp(depth, width, epochs=10):
    model = build_mlp(depth=depth, width=width)
    start_time = time.time()
    history = model.fit(
            X_train, y_train,
            validation_data=(X_val, y_val),
            epochs=epochs,
            verbose=1
        )
    end_time = time.time()
    # Training time per epoch
    time_per_epoch = (end_time - start_time) / epochs
    # Total number of parameters
    total_params = model.count_params()
    return model, history, time_per_epoch, total_params
```

Figure 3: Function to build and train a dynamic MLP. Depth controls the number of hidden layers, width controls neurons per layer, hidden layers use ReLU activation, and the output layer uses softmax for multi-class classification.

5 Results and Analysis

In this section, we present the empirical findings from our depth and width experiments and analyze how increasing model size affects the behavior of Multi-Layer Perceptrons (MLPs). Our results highlight practical limitations of scaling up MLPs because deeper networks suffer from vanishing gradients and slow optimization while wider networks are prone to overfitting despite achieving lower training loss. We also examine how depth and width influence the computational cost of training by comparing the relationship between parameter count and training time per epoch. Together, these experiments illustrate why simply making an MLP larger does not guarantee better performance, rather in many cases it hinders both optimization and generalization.

5.1 Effect of Increasing Depth on Training (Vanishing Gradients)

Figures 4 and 5 summarize how network depth influences training behavior. As the number of hidden layers increases, the training dynamics exhibit clear signs of the vanishing gradient problem.

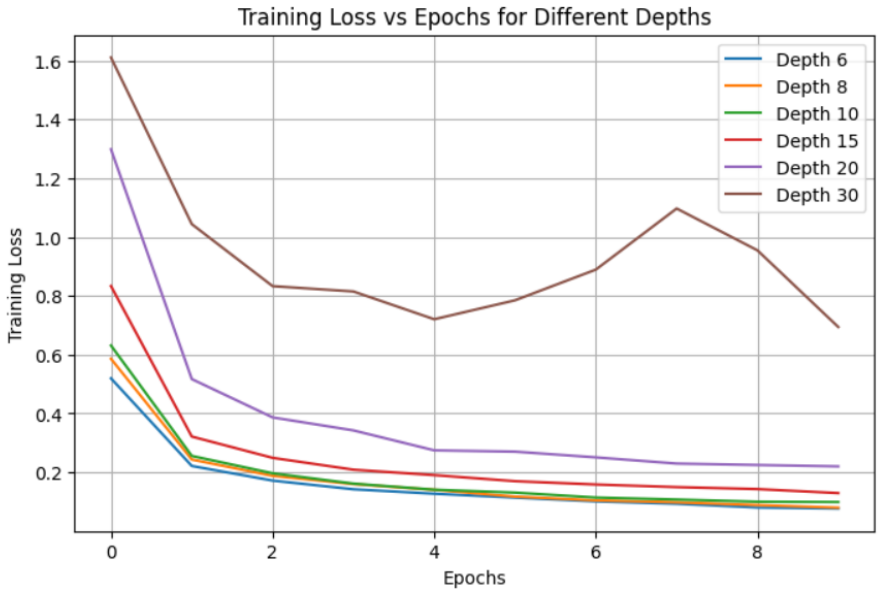

Figure 4: Training loss curves for networks of varying depth.

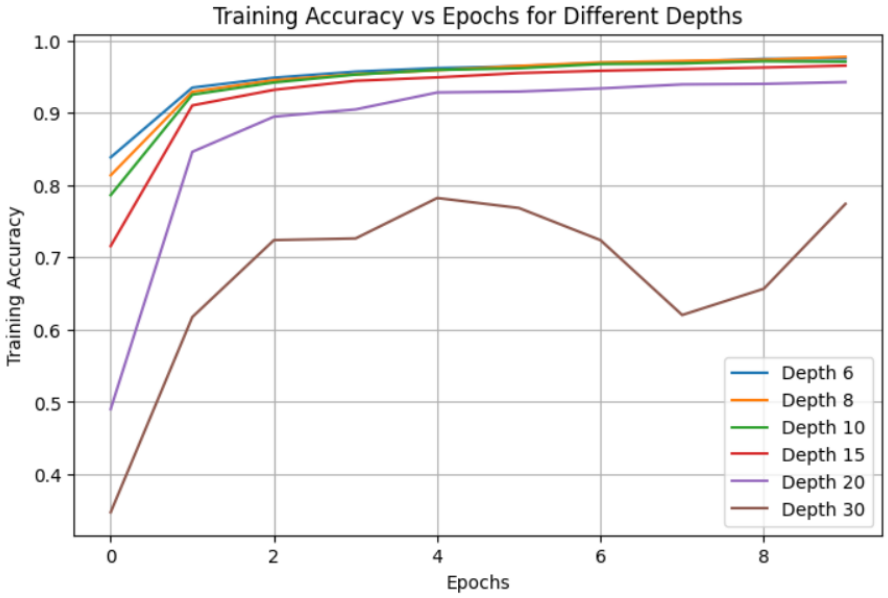

Figure 5: Training accuracy as a function of network depth.

As the depth increases, the gradients propagated back to earlier layers become progressively smaller. This causes the initial layers to update extremely slowly, even when later layers continue to adapt. The training loss curves in Figure 4 illustrate this effect: shallow models converge steadily, whereas deeper models show stagnation or very slow improvement despite extended training. A similar trend appears in training accuracy (Figure 5). Networks with moderate depth achieve reasonable generalization, but extremely deep architectures struggle to match their performance. This degradation occurs not because additional layers reduce the representational capacity, but because the optimization process fails to effectively update the parameters of the earlier layers. Overall, the results demonstrate that simply stacking more layers does not guarantee improved performance; without mechanisms such as residual connec-

tions, careful initialization, or normalization techniques, deep networks are highly susceptible to vanishing gradients during training.

5.2 Effect of Increasing Width on Training (Overfitting)

To examine how network width influences generalization, we trained models with progressively larger numbers of neurons per hidden layer. Figures 6 and 7 present the resulting training and validation performance trends.

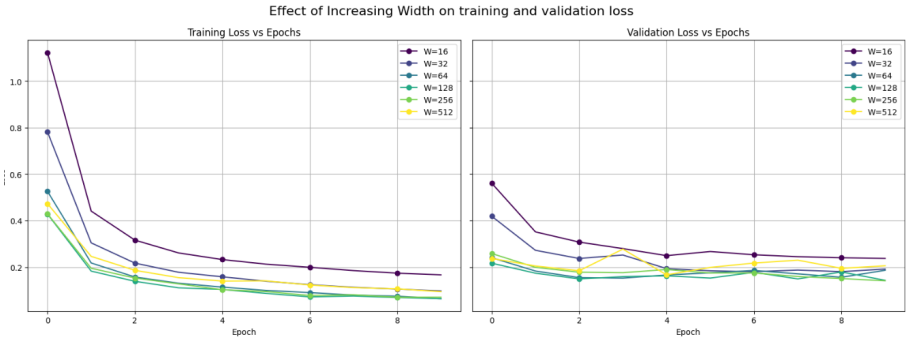

Figure 6: Training loss and validation loss as a function of network width.

Figure 7: Validation accuracy and training accuracy as a function of network width.

Increasing the width of the network consistently reduces training loss, indicating that wider architectures have greater capacity to fit the training data. However, Figure 6 shows a widening gap between training and validation loss as width grows. This behavior is characteristic of overfitting: the model begins to memorize specific patterns in the training set rather than learning generalizable features. The validation accuracy curves in Figure 7 reinforce this observation. While moderately wide networks achieve solid performance, excessively wide models display a notable decline in validation accuracy, even though their training accuracy remains near perfect. This discrepancy highlights that additional capacity does not automatically lead to improved generalization; instead, it can cause the model to latch onto noise or irrelevant structure in the training data. Overall, these results show that wider networks must be paired with suitable regularization techniques—such as dropout, weight decay, or early stopping—to avoid severe overfitting and maintain strong performance on unseen data.

5.3 Training Difficulty of MLPs (Computational Cost)

Training efficiency and stability are strongly influenced by both the depth and width of a Multi-Layer Perceptron. Figure 8 directly compares the Training Time per Epoch against the Total Number of Parameters for both experiments.

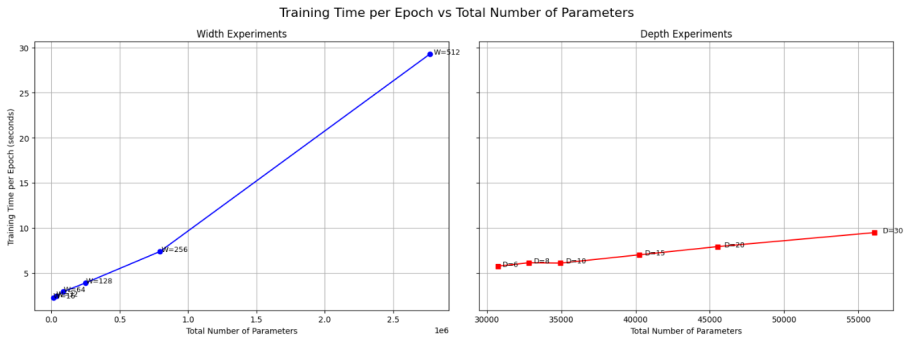

Figure 8: Training time per epoch as a function of the total number of parameters.

Two major factors contribute to the training difficulty and cost of MLPs, as shown in Figure 8:

- **Impact of Width (Computational Cost):** The width experiments plot (left, blue line) demonstrates a near-linear relationship between the training time per epoch and the total number of parameters. As the width (W) of the hidden layers increases, the number of parameters grows significantly, leading to a massive increase in computation required for both the forward and backward passes. For instance, the model with $W = 512$ has a parameter count exceeding 2.5 million and takes substantially longer per epoch, confirming that width is the dominant factor determining the computational cost of training an epoch.
- **Impact of Depth (Optimization Difficulty):** The depth experiments plot (right, red line) shows that varying the depth (D) from 2 to 30 layers results in only a minor increase in the total number of parameters. Consequently, the training time per epoch increases only slightly compared to the width experiment. This shows that increasing depth is computationally cheaper than increasing width when the width is fixed. However, as noted in Section 5.1, as networks become deeper, the optimization becomes challenging due to the vanishing gradient problem, leading to sluggish convergence and poor final performance, despite the lower computational cost per epoch.

These observations support earlier findings in deep learning literature that increasing depth introduces gradient propagation challenges [2, 1], while excessive width increases the risk of memorization [3].

References

[1] Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," **Proceedings of AISTATS**, 2010.

[2] Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," **IEEE Transactions on Neural Networks**, vol. 5, no. 2, 1994.

[3] Zhang, Chiyuan, et al. "Understanding deep learning requires rethinking generalization." arXiv preprint arXiv:1611.03530 (2016).

[4] Goodfellow, O. Vinyals, and A. Saxe, "Qualitatively characterizing neural network optimization problems," arXiv:1412.6544, 2014.