01-basics_neural_networks

April 11, 2025

1 1. Introduction to Neural Networks via Regression

In the previous chapter, we introduced **linear regression**, where we found a model that describes a target variable y using a set of features x_i. In its simplest form, a linear model is defined as:

$$y = w_0 x_0 + w_1 x_1 + \cdots + b$$

We also learned how to optimize the parameters w_i and b by minimizing the prediction error using **gradient descent**, which adjusts parameters in the direction of the steepest descent of the loss function.

In this chapter, we will reformulate this problem using a **neural network**. While neural networks are widely used in deep learning and can become quite complex, their fundamental structure is relatively simple. Linear regression serves as an excellent entry point to understand how neural networks work.

1.1 1.1 What Is a Neural Network?

Neural networks are inspired by early models of the brain. Biological neurons are specialized cells connected to each other through structures called axons, forming vast networks of interconnected cells. These connections can strengthen with experience, enabling learning.

In computational models, we simplify this concept. A **neural network** consists of artificial neurons (nodes) connected by weighted links. Each connection has a **weight** that determines its strength. The basic building block of a neural network resembles the following:

$$y = w_0 x_0 + w_1 x_1 + w_2 x_2 + b$$

This is the same equation we used in linear regression.

Below, we show a an illustration of two connected neurons (left) and a network of cortical neurons of a mouse (right; source: ALol88, CC BY 4.0, via Wikimedia Commons):

```
<div style="flex: 1; text-align: center;">
    <img src="../illustrations/two_connected_neurons.png" style="max-width: 95%; height: auto;"
</div>
<div style="flex: 1; text-align: center;">
    <img src="../illustrations/neurons.jpg" style="max-width: 95%; height: auto;"/>
</div>
```

Neurons communicate through electrical signals, and the strength of their connections can change with experience. If two neurons are frequently activated together, their connection tends to become stronger. These biological networks are vast and highly interconnected.

Artificial neural networks mimic this idea. They consist of:

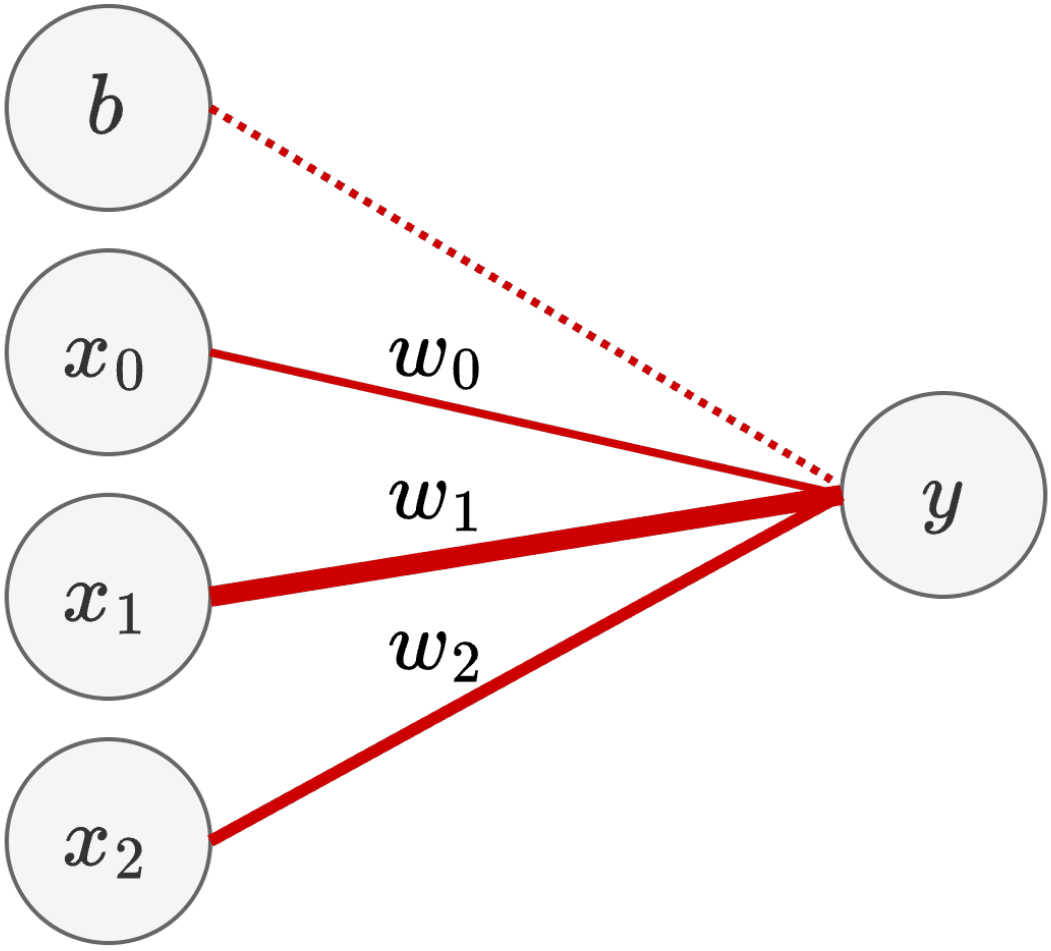
- **Nodes (neurons)** that represent inputs or internal activations,
- **Connections (weights)** that determine how strongly one neuron influences another.

In the simplest case, a neural network can represent the same model as linear regression. The following diagram shows such a minimal network, where the output is computed as:

$$y = w_0 x_0 + w_1 x_1 + w_2 x_2 + b$$

[45]:
```
from IPython.display import Image

Image("../illustrations/networks1.png", width=300)
```

[45]:



The neural network above is the simplest possible configuration. It directly corresponds to the linear regression model introduced earlier:

$$y = w_0 x_0 + w_1 x_1 + w_2 x_2 + b$$

In this representation:

- x_0, x_1, x_2 are the input features,
- w_0, w_1, w_2 are the weights (or parameters),
- b is the bias (intercept),
- y is the predicted output.

This example shows that a basic neural network can implement a linear model. As we will see, we can build more powerful models by adding layers and introducing non-linearities.

1.2 1.2 Tensors

To implement a neural network in code, we need a data structure that can represent and manipulate multi-dimensional arrays. While we could use NumPy arrays for basic operations, deep learning frameworks rely on a more flexible structure called a **tensor**.

Tensors are similar to NumPy arrays but come with additional features:

1. They support computation on **Graphical Processing Units (GPUs)** for faster training.
2. They integrate with **automatic differentiation**, which is essential for gradient-based optimization.

In this notebook, we use PyTorch, which provides a `torch.Tensor` object with NumPy-like syntax and deep learning capabilities.

[46]:
```
import torch

# Create a 3x4 tensor filled with zeros
torch_array = torch.zeros((3, 4))
torch_array
```

[46]:
```
tensor([[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]])
```

We created a 3×4 tensor filled with zeros. This behaves very similarly to a NumPy array.

Many functions in NumPy have equivalents in PyTorch. For example, `torch.zeros` is analogous to `np.zeros`. The key difference is that PyTorch tensors can be used in deep learning pipelines, optimized on GPUs, and integrated into automatic differentiation workflows.

1.3 1.3 Layers

In the neural network diagram shown earlier, the values x_0, x_1, x_2 form the **input layer**. A layer is simply a group of neurons (or units) that process inputs in parallel.

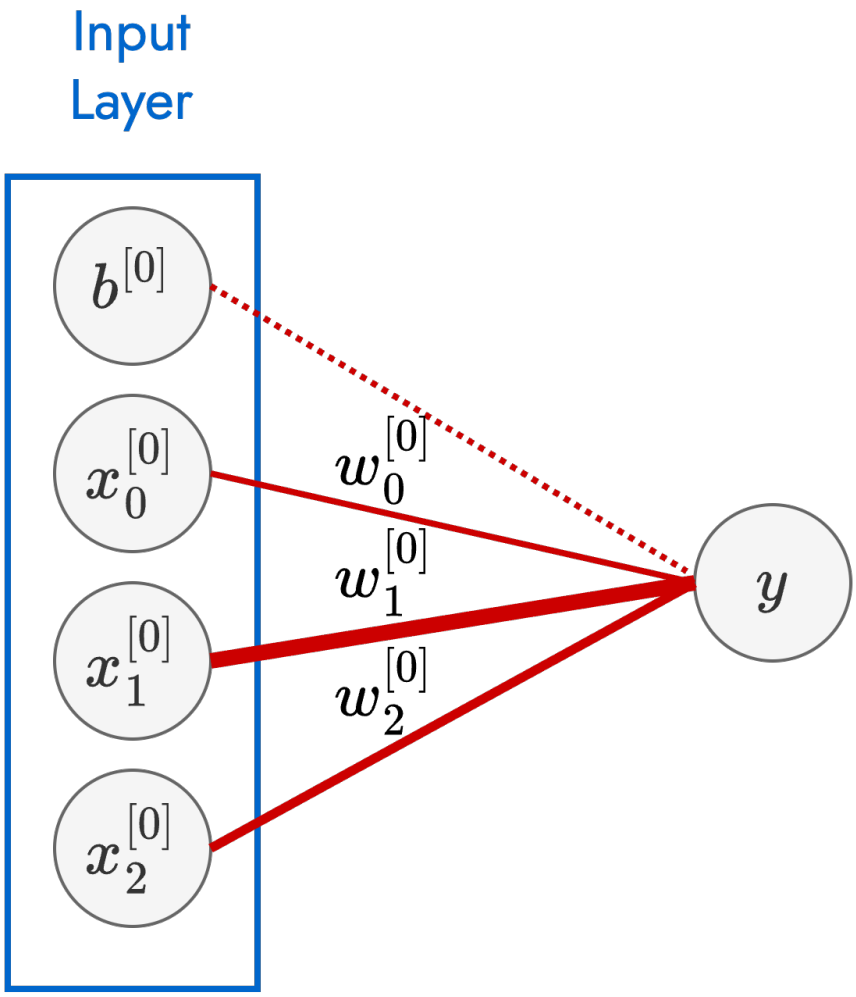
Each neuron computes a weighted sum of its inputs and optionally adds a bias. For example:

$$y = w_0 x_0 + w_1 x_1 + w_2 x_2 + b$$

This corresponds to a **fully connected layer** with three inputs and one output.

[47]:
```
Image("../illustrations/networks2.png", width=500)
```

[47]:



$$y = w_0^{[0]} x_0^{[0]} + w_1^{[0]} x_1^{[0]} + w_2^{[0]} x_2^{[0]} + b^{[0]}$$

We don't need to manually define every weight and bias. Instead, we use PyTorch's `nn` module, which provides predefined layer types like `nn.Linear`, representing a fully connected (dense) layer.

[48]:
```
from torch import nn

# Create a linear layer with 3 input features and 1 output
lin_layer = nn.Linear(in_features=3, out_features=1)
lin_layer
```

[48]:
```
Linear(in_features=3, out_features=1, bias=True)
```

This layer takes a vector of 3 input features and produces a single output. Internally, it maintains:

- Three weights, one for each input feature,
- One bias term, which acts as an intercept.

We can inspect the parameters of the layer as follows:

[49]:
```
# Show the parameters of the linear layer
list(lin_layer.parameters())
```

[49]:
```
[Parameter containing:
 tensor([[-0.1450,  0.2297,  0.0817]], requires_grad=True),
 Parameter containing:
 tensor([0.1585], requires_grad=True)]
```

The output consists of two tensors:

1. A 1×3 weight tensor (since we have 3 inputs and 1 output),
2. A single bias term.

These parameters are initialized randomly. Notice that each parameter has `requires_grad=True`, which means PyTorch will compute gradients with respect to them during backpropagation. This is essential for learning.

Next, we wrap this layer in a full model using `nn.Sequential`, which allows us to stack multiple layers into a network.

[50]:
```
# Wrap the layer in a Sequential model
model = nn.Sequential(lin_layer)
model
```

[50]:
```
Sequential(
  (0): Linear(in_features=3, out_features=1, bias=True)
)
```

We have now defined a simple model consisting of a single linear layer. This model behaves exactly like a linear regression function.

To make a prediction, we provide input values for the features x_0, x_1, x_2. These values need to be passed as a PyTorch tensor with the correct data type.

[51]:
```
# Create an input tensor for prediction
input_tensor = torch.tensor([3, 2, 5], dtype=torch.float32)
input_tensor
```

[51]:
```
tensor([3., 2., 5.])
```

[52]:
```
# Pass the input through the model to get a prediction
model(input_tensor)
```

[52]:
```
tensor([0.5916], grad_fn=<ViewBackward0>)
```

The model returns a prediction based on the current (randomly initialized) weights and bias.

Since the model has not been trained yet, the output is arbitrary. In the next section, we will walk through how to optimize these parameters so that the model fits real data.

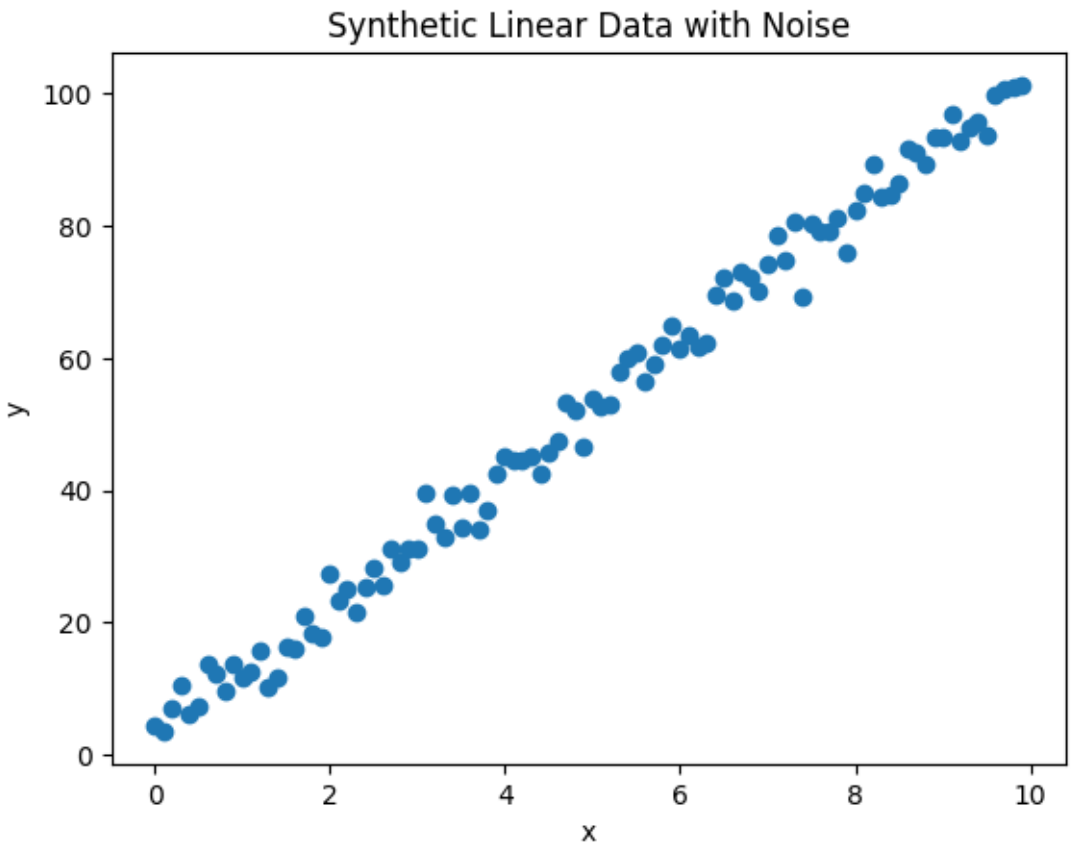
1.4 1.4 Practical Example

Let us now train the model to fit a simple dataset. We will generate synthetic data that follows a linear relationship with some added noise:

$$y = w_0 x + b + \text{noise}$$

Our goal is to learn the parameters w_0 and b from this data using gradient descent.

[53]:
```
import numpy as np
import torch
import matplotlib.pyplot as plt

# Set seeds for reproducibility
np.random.seed(42)
torch.manual_seed(42)

# Generate input values
x_val = torch.arange(0, 10, 0.1)

# Generate corresponding output values with noise
y_val = 3 + 10 * x_val + 3 * torch.tensor(np.random.randn(len(x_val)),␣
 ↪dtype=torch.float32)

# Visualize the data
plt.plot(x_val, y_val, "o")
plt.xlabel("x")
plt.ylabel("y")
plt.title("Synthetic Linear Data with Noise")
plt.show()
```



We created a dataset where the true underlying relationship is:

$$y = 3 + 10 \cdot x + \text{noise}$$

This means the true parameters are $w_0 = 10$ and $b = 3$, but we have added some random noise to make the learning problem more realistic.

We will now define a new linear model and train it to recover these parameters from the noisy data.

[54]:
```
# Define a new linear model with 1 input and 1 output
lin_layer = nn.Linear(in_features=1, out_features=1)

# Show initial parameters (weights and bias)
list(lin_layer.parameters())
```

[54]:
```
[Parameter containing:
 tensor([[0.7645]], requires_grad=True),
 Parameter containing:
 tensor([0.8300], requires_grad=True)]
```

[55]:
```
# Wrap the linear layer in a Sequential model
model = nn.Sequential(lin_layer)
```

This model has a single input and a single output. It contains two parameters:

- A weight w_0 for the input feature,
- A bias b (intercept).

Both are initialized randomly. We now need to define two components to train the model:

1. A **loss function** to measure how well the model performs.
2. An **optimizer** that adjusts the parameters based on the loss.

[56]:
```
# Define the loss function: Mean Squared Error (MSE)
criterion = nn.MSELoss()
```

[57]:
```
from torch import optim

# Use stochastic gradient descent (SGD) with a small learning rate
optimizer = optim.SGD(model.parameters(), lr=0.001)
```

[58]:
```
# Take the first input value
inputs = x_val[0]
inputs
```

[58]:
```
tensor(0.)
```

[59]:
```
# Reshape the input to a batch with one sample and one feature
inputs = inputs.unsqueeze(0)
inputs
```

[59]:
```
tensor([0.])
```

[60]:
```
# Compute the model output for the input
outputs = model(inputs)
outputs
```

[60]:
```
tensor([0.8300], grad_fn=<ViewBackward0>)
```

[61]:
```
# Select the corresponding target value and reshape it
targets = y_val[0].unsqueeze(0)
targets
```

[61]:
```
tensor([4.4901])
```

[62]:
```
# Compute the loss (mean squared error) for this single example
loss = criterion(outputs, targets)
loss
```

[62]:
```
tensor(13.3966, grad_fn=<MseLossBackward0>)
```

[63]:
```
# Manually compute the squared error to verify the loss
((targets[0] - outputs[0]) ** 2).item()
```

[63]:
```
13.396583557128906
```

[64]:
```
# Compute gradients via backpropagation
loss.backward()
```

[65]:
```
# Perform one optimization step using the computed gradients
optimizer.step()
```

[66]:
```
# Reset gradients to zero (important before the next backward pass)
optimizer.zero_grad()
```

[67]:
```
# Forward pass after one optimization step
outputs = model(inputs)
loss = criterion(outputs, targets)
loss
```

[67]:
```
tensor(13.3431, grad_fn=<MseLossBackward0>)
```

We can see that the loss has slightly decreased compared to the initial value. This means the model has started to learn.

However, training on a single data point is not sufficient. In practice, we want to use the entire dataset to update the model parameters. PyTorch models can process multiple inputs at once if we format the data correctly — as a matrix where each row is a data point and each column is a feature.

[68]:
```
# Reshape x_val to have shape (num_samples, num_features)
all_inputs = x_val[:, np.newaxis]

# Forward pass for all inputs
outputs = model(all_inputs)
outputs.shape
```

[68]:
```
torch.Size([100, 1])
```

[69]:
```
# Reshape y_val similarly to match output shape
all_targets = y_val[:, np.newaxis]

# Compute the loss over the full dataset
loss = criterion(outputs, all_targets)
loss
```

[69]:
```
tensor(2987.0796, grad_fn=<MseLossBackward0>)
```

1.5 1.5 Repeating the Optimization

To train the model properly, we repeat the following steps many times:

1. Compute predictions from the current model.
2. Calculate the loss (error).
3. Perform backpropagation to compute gradients.
4. Update the model parameters using the optimizer.
5. Reset the gradients.

This process is repeated for many iterations to gradually minimize the loss. Below, we perform this procedure for 1000 steps and print the loss every 100 iterations.

[70]:
```
# Training loop: repeat optimization steps for 1000 epochs
for epoch in range(1000):
    # Reset gradients
    optimizer.zero_grad()

    # Forward pass
    outputs = model(all_inputs)

    # Compute loss
    loss = criterion(outputs, all_targets)

    # Backpropagation
    loss.backward()

    # Parameter update
    optimizer.step()

    # Print loss every 100 epochs
    if epoch % 100 == 0:
        print(f"Epoch {epoch}, Loss: {loss.item():.4f}")
```

```
Epoch 0, Loss: 2987.0796
Epoch 100, Loss: 7.3490
Epoch 200, Loss: 7.3451
Epoch 300, Loss: 7.3441
Epoch 400, Loss: 7.3431
Epoch 500, Loss: 7.3423
Epoch 600, Loss: 7.3415
Epoch 700, Loss: 7.3408
Epoch 800, Loss: 7.3402
Epoch 900, Loss: 7.3396
```

The model starts with a high initial loss around 3000, which reflects the randomly initialized parameters. After training:

- At epoch 100, the loss has dropped below 7.35.
- The loss continues to decrease slowly and steadily, reaching approximately 7.34 after 900 epochs.

This behavior shows that the model is gradually learning the linear relationship between x and y.

[71]:
```
# Predict values with the trained model
pred = model(all_inputs)

# Detach the prediction tensor and convert it to a NumPy array for plotting
pred_np = pred.detach().numpy()

# Plot original data and model predictions
plt.plot(x_val, y_val, "o", label="Noisy Data")
plt.plot(x_val, pred_np, "r", label="Model Prediction")
plt.xlabel("x")
plt.ylabel("y")
plt.title("Model Fit After Training")
plt.legend()
plt.show()
```

The plot shows the noisy data points (in blue) and the model's predictions (in red).

As the model has been trained on this data, we can see that the predictions closely follow the underlying linear trend, despite the noise added to the data. This indicates that the model has

successfully learned the general pattern.

However, since we only used a simple linear model, it might not perfectly fit more complex data. In the next sections, we can explore how adding layers or non-linearities can improve the model's capacity to fit more complicated patterns.

1.6 1.6 Mini-Batches

In the previous sections, we trained the model on the entire dataset at once. While this is fine for small datasets, it is often inefficient for larger datasets. Instead, we can split the data into smaller groups called **mini-batches**. This allows the model to learn from smaller subsets of data at a time, which can speed up training and improve generalization.

Mini-batch training is widely used because it strikes a balance between computational efficiency and model accuracy. In the following section, we will modify our training loop to use mini-batches instead of processing the entire dataset at once.

To clearly illustrate mini-batch training, let's define and initialize a new model from scratch.

[72]:
```
# Define a fresh linear model for mini-batch training
lin_layer_mb = nn.Linear(in_features=1, out_features=1)
model_mb = nn.Sequential(lin_layer_mb)

# Define loss function and optimizer again
criterion_mb = nn.MSELoss()
optimizer_mb = optim.SGD(model_mb.parameters(), lr=0.001)
```

[73]:
```
# Define mini-batch size
batch_size = 10

# Mini-batch training loop
for epoch in range(10):  # limited epochs for demonstration
    for i in range(0, len(all_inputs), batch_size):
        # Select mini-batch
        inputs = all_inputs[i : i + batch_size]
        targets = all_targets[i : i + batch_size]

        # Reset gradients
        optimizer_mb.zero_grad()

        # Forward pass
        outputs = model_mb(inputs)

        # Compute loss
        loss = criterion_mb(outputs, targets)

        # Backpropagation
        loss.backward()
```

```
        # Parameter update
        optimizer_mb.step()

    # Print loss at the end of each epoch
    print(f"Epoch {epoch}, Loss: {loss.item():.4f}")
```

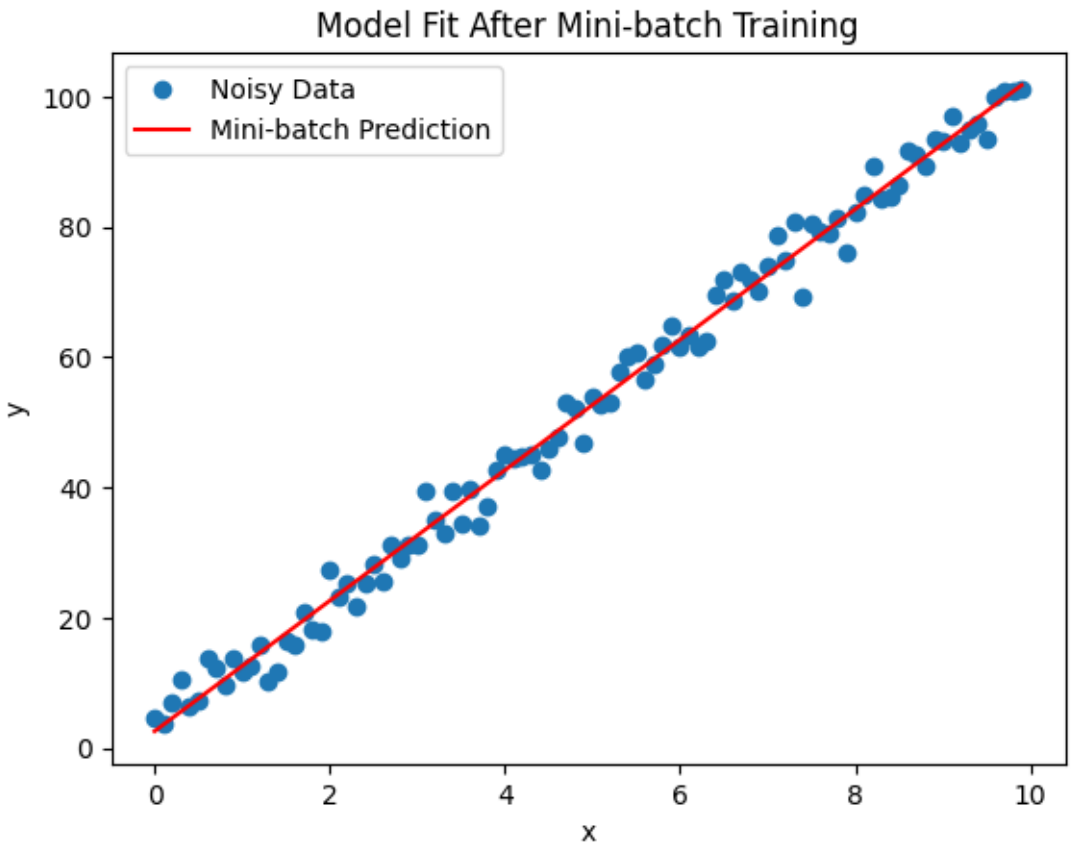
```
Epoch 0, Loss: 3425.1086
Epoch 1, Loss: 809.6439
Epoch 2, Loss: 190.7037
Epoch 3, Loss: 45.6478
Epoch 4, Loss: 12.3499
Epoch 5, Loss: 5.0537
Epoch 6, Loss: 3.6317
Epoch 7, Loss: 3.4488
Epoch 8, Loss: 3.4817
Epoch 9, Loss: 3.5270
```

The mini-batch training output shows that the loss rapidly decreases during the first few epochs:

- The initial loss at epoch 0 is very high (around 3425), reflecting randomly initialized parameters.
- Within just a few epochs, the loss significantly decreases, reaching approximately 12 by epoch 4.
- After epoch 5, improvements become smaller and the loss stabilizes around 3.5.

This demonstrates that training with mini-batches allows the model to update its parameters frequently, leading to rapid initial learning. It also highlights that after a certain point, continuing to train may result in minimal improvements, indicating that the model has reached a stable solution.

[74]:
```
# Generate predictions with the mini-batch trained model
pred_mb = model_mb(all_inputs)

# Convert predictions to numpy for plotting
pred_mb_np = pred_mb.detach().numpy()

# Plot original data and mini-batch trained model predictions
plt.plot(x_val, y_val, "o", label="Noisy Data")
plt.plot(x_val, pred_mb_np, "r", label="Mini-batch Prediction")
plt.xlabel("x")
plt.ylabel("y")
plt.title("Model Fit After Mini-batch Training")
plt.legend()
plt.show()
```



1.7 1.7 Adding Layers

So far, our neural network consisted of just one layer, making it equivalent to simple linear regression. However, neural networks become powerful when we add multiple layers. These additional layers enable the network to capture more complex patterns in the data.

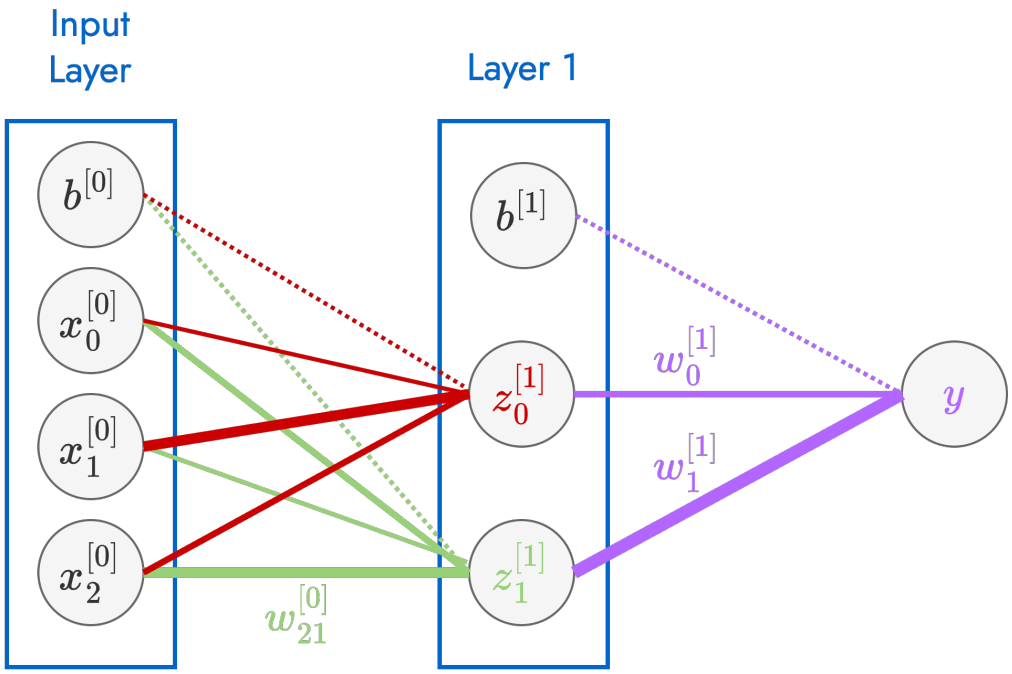
Below is an illustration of a neural network with multiple layers:

- An input layer with three features,
- A hidden layer with two neurons (hidden units),
- An output layer producing the final prediction.

Each neuron in a layer is connected to every neuron in the next layer. By stacking layers, the network can learn hierarchical representations of data.

[75]:
```
Image("../illustrations/networks3.png", width=600)
```

[75]:



$$z_0^{[1]} = w_{00}{}^{[0]} x_0^{[0]} + w_{10}^{[0]} x_1^{[0]} + w_{20}^{[0]} x_2^{[0]} + b^{[0]}$$

$$z_1^{[1]} = w_{01}^{[0]} x_0^{[0]} + w_{11}^{[0]} x_1^{[0]} + w_{21}^{[0]} x_2^{[0]} + b^{[0]}$$

$$y = w_0^{[1]} z_0^{[1]} + w_1^{[1]} z_1^{[1]} + b^{[1]}$$

Let's create a neural network that is slightly more complex than before, with two layers:

[76]:
```
# Define a model with two fully connected layers
lin_layer1 = nn.Linear(in_features=1, out_features=10)
lin_layer2 = nn.Linear(in_features=10, out_features=1)

# Wrap the layers in a Sequential container
model_2layer = nn.Sequential(lin_layer1, lin_layer2)

# Display model architecture
model_2layer
```

[76]:
```
Sequential(
  (0): Linear(in_features=1, out_features=10, bias=True)
  (1): Linear(in_features=10, out_features=1, bias=True)
```

)

This model consists of two fully connected layers:

- The first layer takes a single input and produces 10 outputs (hidden units).
- The second layer takes these 10 hidden units as input and produces a single output.

Even though this network is deeper than the previous one, it is still a **linear model** because it consists only of linear transformations stacked together. The composition of linear functions is still linear.

To model more complex data patterns, we need to introduce **non-linear activation functions** between the layers.

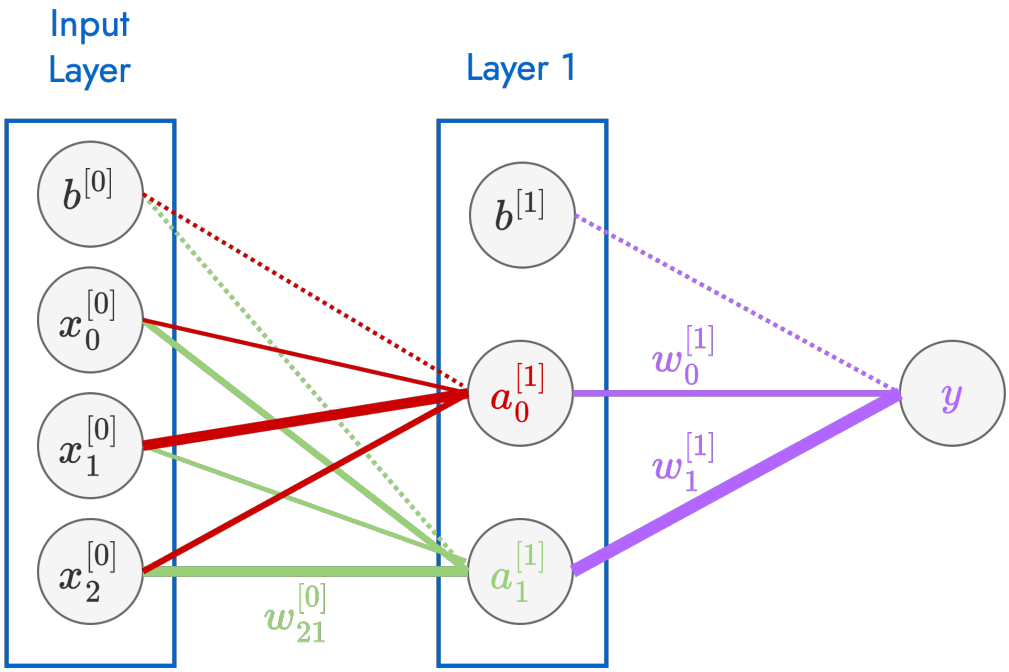
1.8 1.8 Adding Non-Linearities: Activation Functions

In many real-world problems, the relationship between inputs and outputs is not linear. Stacking multiple linear layers cannot help in this case, because their composition is still a linear function.

To allow the network to learn **non-linear patterns**, we insert a non-linear **activation function** between layers. This function is applied element-wise to the outputs of a layer before passing them to the next layer.

[77]:
```
Image("../illustrations/networks4.png", width=600)
```

[77]:



$$a_0^{[1]} = g(z_0^{[1]}) = g(w_{00}{}^{[0]} x_0^{[0]} + w_{10}^{[0]} x_1^{[0]} + w_{20}^{[0]} x_2^{[0]} + b^{[0]})$$

$$a_1^{[1]} = g(z_1^{[1]}) = g(w_{01}^{[0]} x_0^{[0]} + w_{11}^{[0]} x_1^{[0]} + w_{21}^{[0]} x_2^{[0]} + b^{[0]})$$

$$y = w_0^{[1]} z_0^{[1]} + w_1^{[1]} z_1^{[1]} + b^{[1]}$$

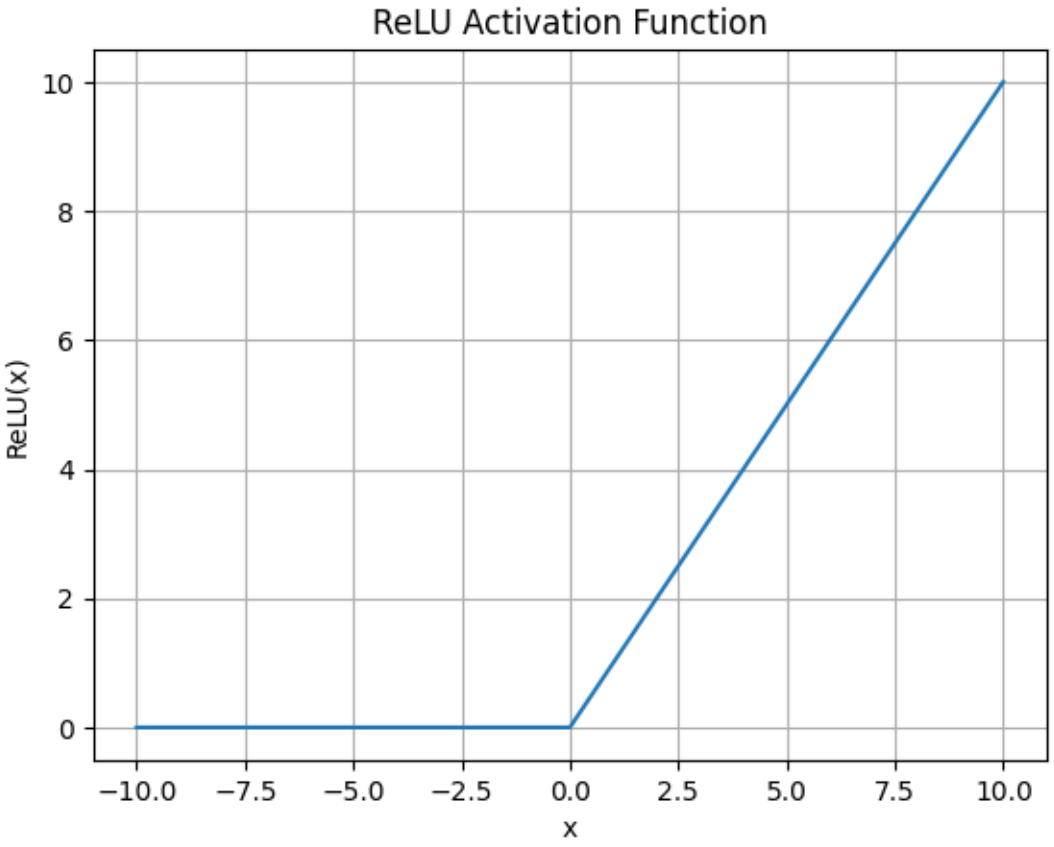
A commonly used activation function is the **Rectified Linear Unit (ReLU)**, defined as:

$$\text{ReLU}(x) = \max(0, x)$$

It outputs the input directly if it is positive, and zero otherwise. Despite its simplicity, ReLU allows neural networks to model complex, non-linear relationships effectively.

[78]:
```
from torch.nn.functional import relu

# Create input values ranging from -10 to 10
x = torch.arange(-10.0, 10.1, 0.1)

# Apply the ReLU activation
y = relu(x)
```

```
# Plot the ReLU function
fig, ax = plt.subplots()
ax.plot(x, y)
ax.set_title("ReLU Activation Function")
ax.set_xlabel("x")
ax.set_ylabel("ReLU(x)")
plt.grid(True)
plt.show()
```

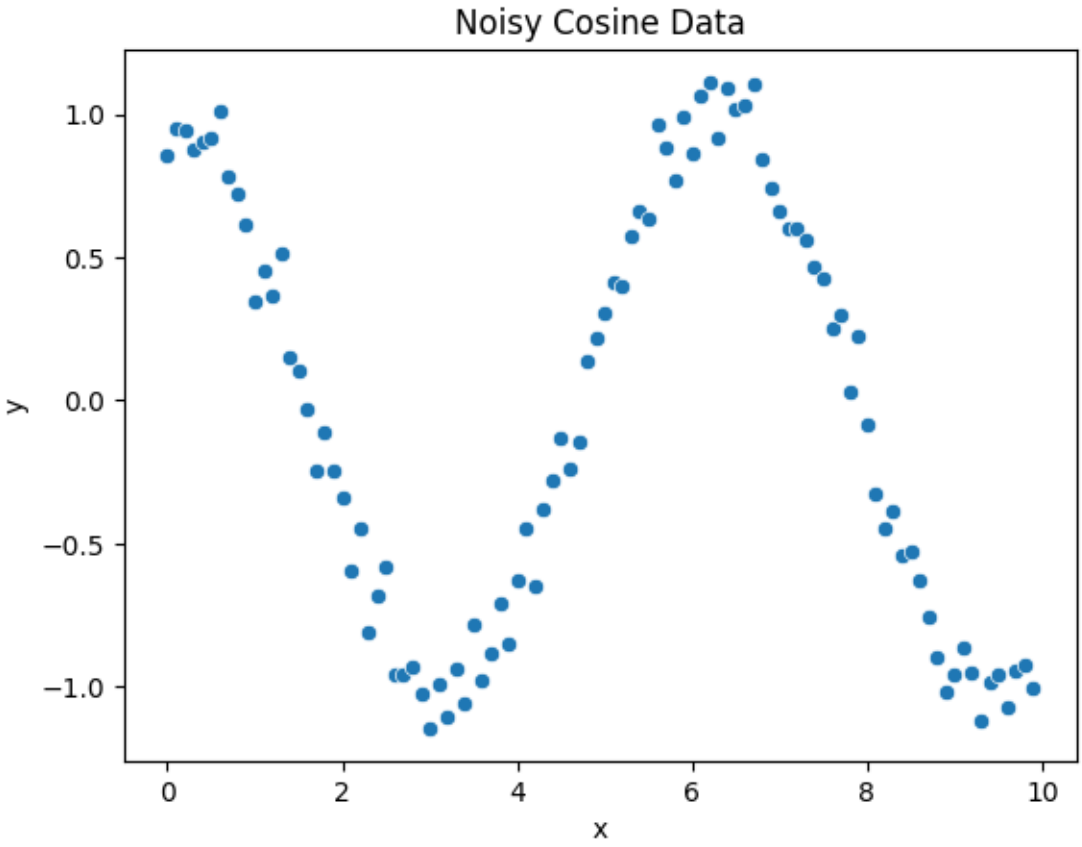


Let us now apply a neural network with ReLU activations to a non-linear dataset.

We will generate data that follows a cosine function with added noise. This kind of data cannot be modeled accurately by a simple linear regression model. By stacking multiple layers with activation functions, we give the network the capacity to approximate this non-linear relationship.

[79]:
```
import seaborn as sns

# Generate a non-linear dataset: cosine function with noise
x_val = torch.arange(0, 10, 0.1)
```

```
y_val = torch.cos(x_val) + 0.1 * torch.tensor(np.random.randn(len(x_val)),␣
 ↪dtype=torch.float32)

# Visualize the data
sns.scatterplot(x=x_val, y=y_val)
plt.xlabel("x")
plt.ylabel("y")
plt.title("Noisy Cosine Data")
plt.show()
```



To model this non-linear data, we will define a deeper neural network with multiple layers and insert ReLU activation functions between them.

The network will consist of:

- An input layer with 1 feature,
- Two hidden layers with 64 units each, using ReLU activations,
- An output layer with 1 unit to predict the target value.

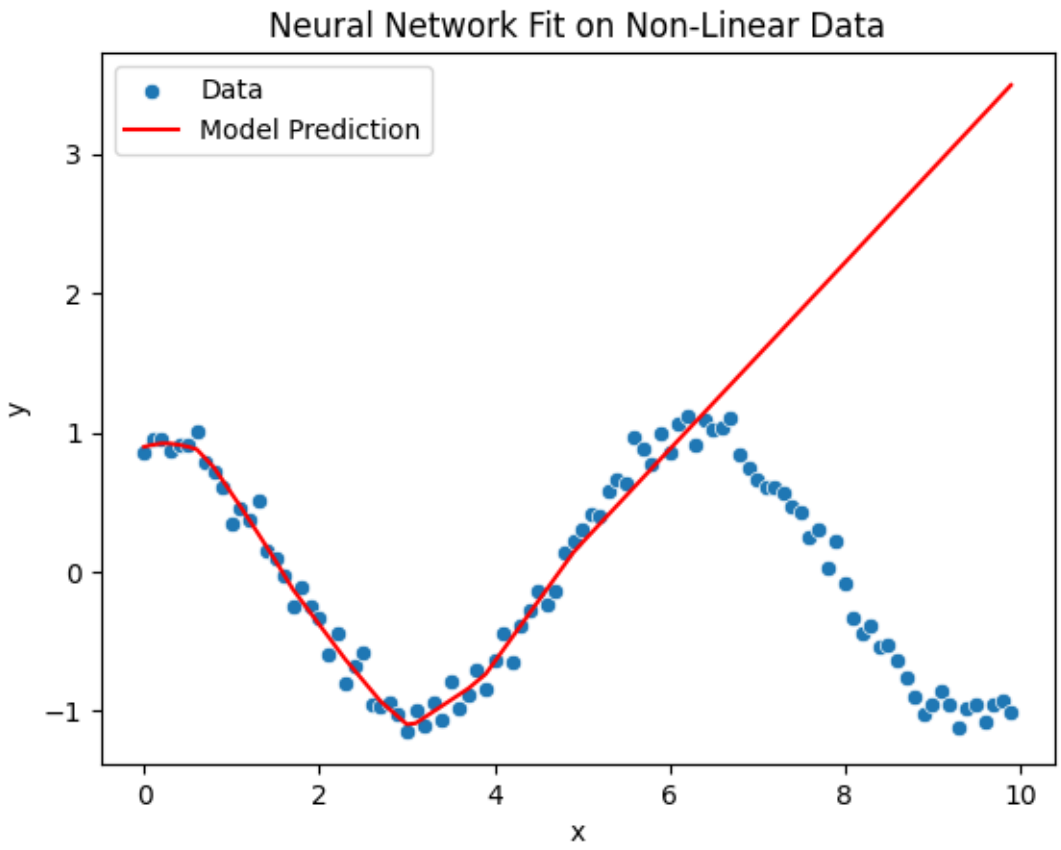
This architecture allows the model to approximate complex non-linear functions without the need for manual feature engineering.

[80]:
```
# Define a deeper neural network with ReLU activations
lin_layer1 = nn.Linear(in_features=1, out_features=64)
lin_layer2 = nn.Linear(in_features=64, out_features=64)
lin_layer3 = nn.Linear(in_features=64, out_features=1)

# Combine layers with activations using Sequential
model_nonlinear = nn.Sequential(lin_layer1, nn.ReLU(), lin_layer2, nn.ReLU(),␣
 ↪lin_layer3)
```

[81]:
```
# Define loss function and optimizer for the non-linear model
criterion = nn.MSELoss()
optimizer = optim.SGD(model_nonlinear.parameters(), lr=0.001)
```

[82]:
```
# Train on the first half of the data (50 points), using mini-batches of size 10
for epoch in range(10000):
    for i in range(5):  # 5 batches of 10 samples each
        # Select inputs and targets for the batch
        inputs = x_val[i * 10 : (i + 1) * 10].unsqueeze(1)
        targets = y_val[i * 10 : (i + 1) * 10].unsqueeze(1)

        # Reset gradients
        optimizer.zero_grad()

        # Forward pass
        outputs = model_nonlinear(inputs)

        # Compute loss
        loss = criterion(outputs, targets)

        # Backward pass and parameter update
        loss.backward()
        optimizer.step()
```

[83]:
```
# Generate predictions over the full dataset
predicted_torch = model_nonlinear(x_val.unsqueeze(1))
pred = predicted_torch.detach().numpy().ravel()

# Plot original data and model prediction
sns.scatterplot(x=x_val, y=y_val, label="Data")
plt.plot(x_val, pred, "r-", label="Model Prediction")
plt.xlabel("x")
plt.ylabel("y")
plt.title("Neural Network Fit on Non-Linear Data")
plt.legend()
plt.show()
```



The plot shows that the neural network is able to fit the first half of the data very well. Since the model was only trained on the first 50 points, it generalizes poorly to the second half of the data, where the prediction diverges.

This behavior highlights an important limitation of deep learning models: they tend to **memorize** the training data and may not generalize well to unseen inputs unless trained carefully. In contrast to hand-crafted models with strong assumptions (such as periodicity), neural networks rely purely on data-driven learning.

1.9 1.9 Overfitting

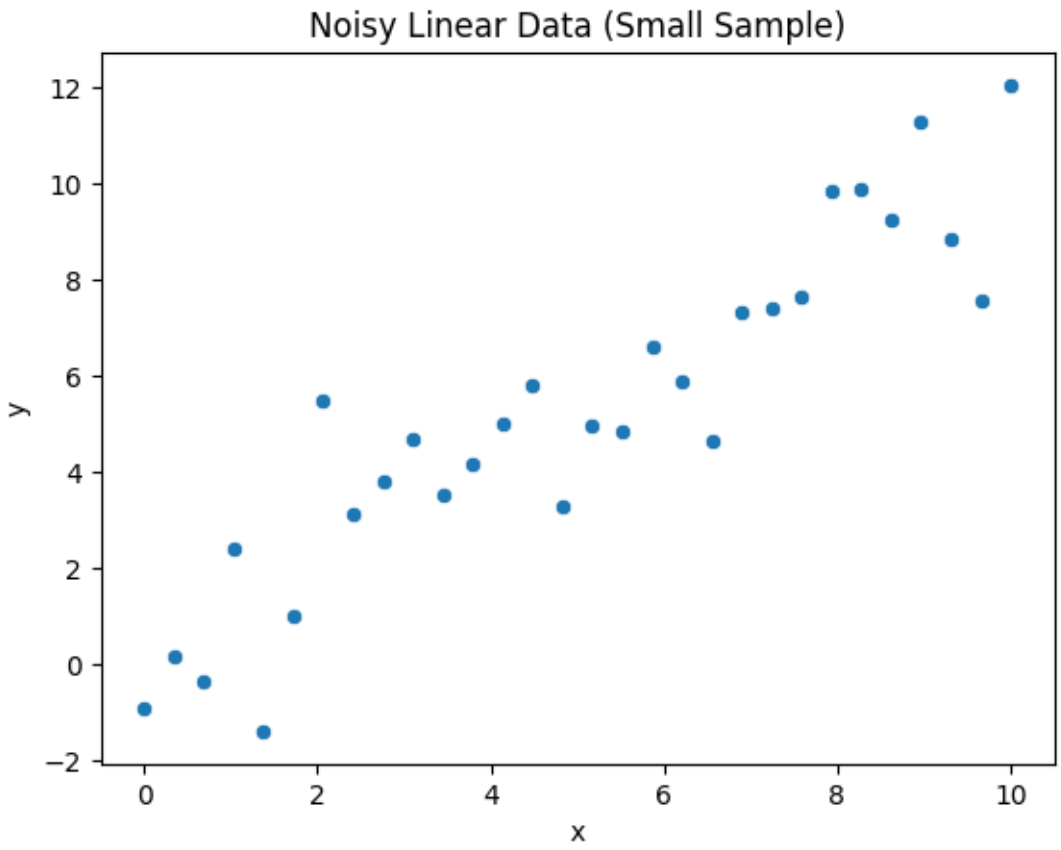
A common issue in machine learning, especially with neural networks, is **overfitting**. Overfitting occurs when a model learns the training data too well, including its noise or random fluctuations, at the cost of generalizing poorly to new data.

This problem becomes more likely when:

- The model is very complex (many layers or parameters),
- The training dataset is small,
- Training continues for too many epochs without regularization.

To monitor for overfitting, we typically evaluate model performance not only on the training data but also on a separate **test set** that the model never sees during training.

[84]:
```
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
import pandas as pd

# Set seed for reproducibility
np.random.seed(40)
torch.manual_seed(40)

# Create small noisy linear dataset
x_val = torch.linspace(0, 10, 30)
y_val = 1 * x_val + 1.5 * torch.tensor(np.random.randn(len(x_val)), dtype=torch.
 ↪float32)

# Wrap in a DataFrame
dataset = pd.DataFrame({"x_val": x_val, "y_val": y_val})

# Split into training and testing sets
dataset_train, dataset_test = train_test_split(dataset, test_size=0.32,␣
 ↪random_state=42)

# Visualize the data
sns.scatterplot(x=dataset["x_val"], y=dataset["y_val"])
plt.xlabel("x")
plt.ylabel("y")
plt.title("Noisy Linear Data (Small Sample)")
plt.show()
```

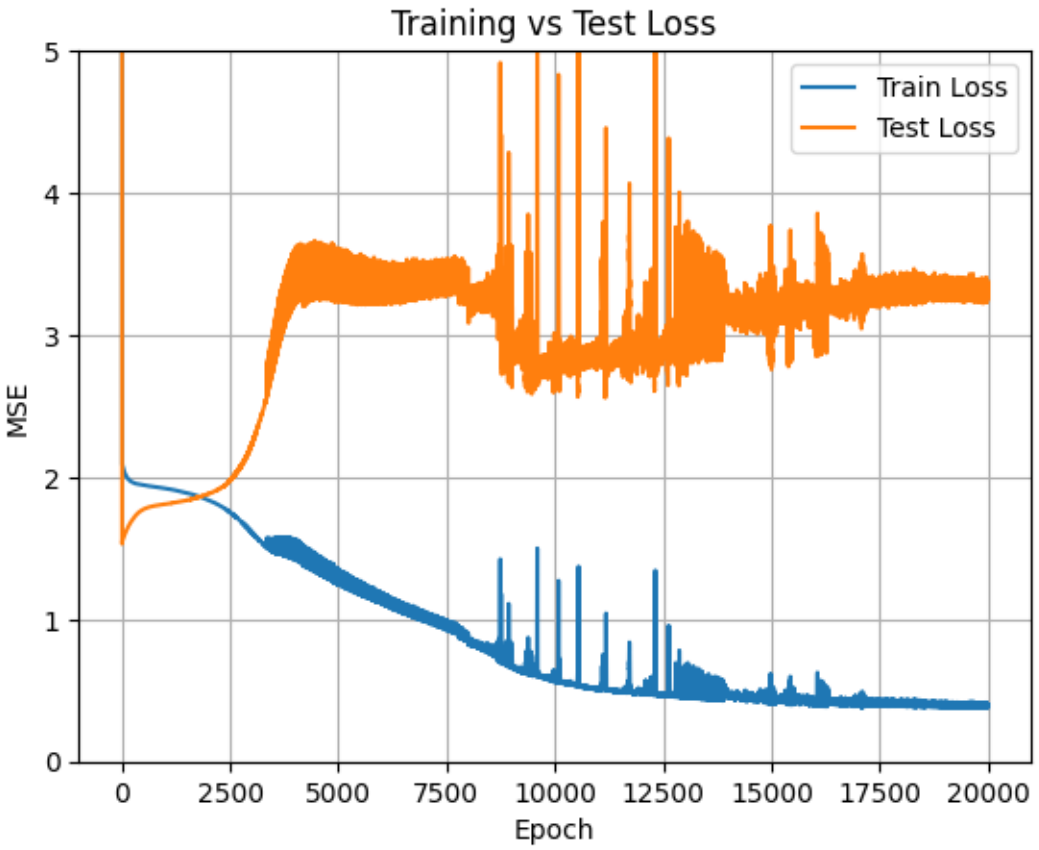


[85]:
```
# Define a large model for a small dataset
lin_layer1 = nn.Linear(in_features=1, out_features=64)
lin_layer2 = nn.Linear(in_features=64, out_features=64)
lin_layer3 = nn.Linear(in_features=64, out_features=64)
lin_layer4 = nn.Linear(in_features=64, out_features=1)

# Stack layers with ReLU activations
model_overfit = nn.Sequential(lin_layer1, nn.ReLU(), lin_layer2, nn.ReLU(),␣
 ↪lin_layer3, nn.ReLU(), lin_layer4)

# Define loss and optimizer
criterion = nn.MSELoss()
optimizer = optim.SGD(model_overfit.parameters(), lr=0.001)
```

[86]:
```
errors_train = []
errors_test = []

# Convert training and testing data to tensors
x_train = torch.tensor(dataset_train[["x_val"]].values, dtype=torch.float32)
```

```
y_train = torch.tensor(dataset_train[["y_val"]].values, dtype=torch.float32)
x_test = torch.tensor(dataset_test[["x_val"]].values, dtype=torch.float32)
y_test = torch.tensor(dataset_test[["y_val"]].values, dtype=torch.float32)

# Train for many epochs to demonstrate overfitting
for epoch in range(20000):
    for i in range(0, len(x_train), 10):
        inputs = x_train[i : i + 10]
        targets = y_train[i : i + 10]

        optimizer.zero_grad()
        outputs = model_overfit(inputs)
        loss = criterion(outputs, targets)
        loss.backward()
        optimizer.step()

    # Evaluate model on train and test sets
    with torch.no_grad():
        train_pred = model_overfit(x_train)
        test_pred = model_overfit(x_test)
        train_loss = mean_squared_error(y_train.numpy(), train_pred.numpy())
        test_loss = mean_squared_error(y_test.numpy(), test_pred.numpy())
        errors_train.append(train_loss)
        errors_test.append(test_loss)
```

[87]:
```
# Plot training and test loss over time
fig, ax = plt.subplots()
ax.plot(errors_train, label="Train Loss")
ax.plot(errors_test, label="Test Loss")
ax.set_ylim(0, 5)
ax.set_xlabel("Epoch")
ax.set_ylabel("MSE")
ax.set_title("Training vs Test Loss")
ax.legend()
plt.grid(True)
plt.show()
```



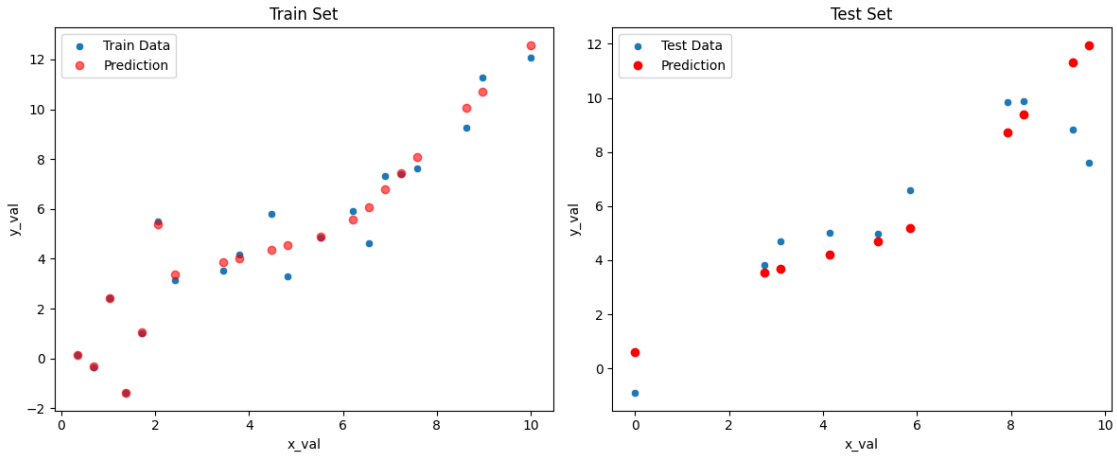
The plot illustrates a typical case of overfitting:

- In the early stages of training, both training and test losses decrease, indicating that the model is learning a useful pattern.
- After some point, the **training loss continues to decrease**, but the **test loss starts to increase**. This means the model is starting to memorize the training data, including its noise, rather than learning a generalizable pattern.

The increasing test loss is a clear sign that the model is **overfitting**. It performs well on data it has seen but generalizes poorly to new inputs.

[88]:
```
# Generate predictions
pred_train = model_overfit(x_train).detach().numpy().ravel()
pred_test = model_overfit(x_test).detach().numpy().ravel()

# Plot predictions
fig, ax = plt.subplots(1, 2, figsize=(12, 5))

# Training set
sns.scatterplot(x=dataset_train["x_val"], y=dataset_train["y_val"], ax=ax[0],␣
↪label="Train Data")
```

```
ax[0].plot(dataset_train["x_val"], pred_train, "ro", alpha=0.6,␣
 ↪label="Prediction")
ax[0].set_title("Train Set")
ax[0].legend()

# Test set
sns.scatterplot(x=dataset_test["x_val"], y=dataset_test["y_val"], ax=ax[1],␣
 ↪label="Test Data")
ax[1].plot(dataset_test["x_val"], pred_test, "ro", label="Prediction")
ax[1].set_title("Test Set")
ax[1].legend()

plt.tight_layout()
plt.show()
```



The plots show that the model fits the training data very closely, even matching noisy fluctuations. On the test set, however, the predictions are far less accurate and deviate significantly from the actual data points.

This confirms that the model has overfitted: it learned the training data too precisely but failed to generalize to new, unseen examples.

In practice, overfitting can be reduced through techniques such as:

- Using more training data,
- Reducing model complexity,
- Adding regularization (e.g., weight decay, dropout),
- Stopping training early based on validation performance.

Understanding overfitting is essential for building models that perform well in real-world settings.

1.10 1.10 Exercise: Predict Insurance Cost from Age

In this exercise, you'll build a very simple neural network to predict a person's **medical insurance charges** based on their **age**.

Your task is to:

- Filter the dataset so that it only contains non-smokers
- Use age as the only input feature
- Predict the insurance cost (charges)
- Build a small neural network (e.g., one hidden layer)
- Train the model and visualize the results

This is a regression task — the output is a continuous number (insurance cost in USD). Part of the code is already implemented. **You need to fill out the parts that are marked with TODO comments.**

[ ]:
```
import pandas as pd
import torch
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split


# Load dataset
url = "https://raw.githubusercontent.com/stedy/Machine-Learning-with-R-datasets/
 ↪master/insurance.csv"
df = pd.read_csv(url)

# Plot data
sns.scatterplot(df, x='age', y='charges', hue='smoker')
plt.xlabel("Age")
plt.ylabel("Insurance Charges (USD)")
plt.title("Insurance Charges vs. Age")
plt.grid(True)
plt.show()
```

[ ]:
```
# TODO: filter the dataset to only include non-smokers
...
```

[ ]:
```
df_train, df_test = train_test_split(df, test_size=0.2, random_state=42)

# Select input and target
X_train = df_train[["age"]]
y_train = df_train[["charges"]]
X_test = df_test[["age"]]
y_test = df_test[["charges"]]
```

```
# Normalize features
feature_scaler = StandardScaler()
X_train = feature_scaler.fit_transform(X_train)
X_test = feature_scaler.transform(X_test)

# Normalize target
target_scaler = StandardScaler()
y_train = target_scaler.fit_transform(y_train)
y_test = target_scaler.transform(y_test)

# Convert to PyTorch tensors
X_train = torch.tensor(X_train, dtype=torch.float32)
X_test = torch.tensor(X_test, dtype=torch.float32)
y_train = torch.tensor(y_train, dtype=torch.float32)
y_test = torch.tensor(y_test, dtype=torch.float32)

print(X_train.shape)
print(y_train.shape)
print(X_test.shape)
print(y_test.shape)
```

[ ]:
```
from torch import nn, optim

# TODO: Define a simple feedforward neural network with a non-linear activation␣
 ↪function
model = ...

# TODO: Define loss function (MSE) and optimizer (SGD) with a learning rate of␣
 ↪0.001
criterion = ...
optimizer = ...

# Training loop
for epoch in range(1000):
    # Forward pass
    outputs = model(X_train)

    # TODO: Compute the loss
    loss = ...

    # Set gradients to zero
    optimizer.zero_grad()

    # TODO: Perform a backward pass and an optimization step
    ...

    # Print loss every 100 epochs
```

```
    if (epoch + 1) % 100 == 0:
        print(f"Epoch [{epoch + 1}/1000], Loss: {loss.item():.4f}")
```

[ ]:
```
# Evaluate the model on the test set
with torch.no_grad():
    # TODO: Make predictions on the test set and compute the loss
    y_pred = ...
    test_loss = ...
    print(f"Test Loss: {test_loss.item():.4f}")
```

[ ]:
```
features = feature_scaler.inverse_transform(X_test.numpy())
predictions = target_scaler.inverse_transform(y_pred.numpy())

# TODO: Plot the predictions against the actual values
...
```