Lecture 23

Simple Neural Network

Neural networks are a fundamental component of deep learning, a subset of machine learning. They are inspired by the structure and functioning of the human brain, specifically the way neurons communicate with each other. Neural networks are composed of layers of interconnected nodes (neurons), each layer serving a specific function in transforming the input data into an output.

1. Basic Structure of a Neural Network

A typical neural network consists of three main types of layers:

- *Input Layer*: The input layer is where the network receives its data. Each neuron in this layer represents a feature in the dataset.
- *Hidden Layers*: These layers sit between the input and output layers. They are called "hidden" because they are not directly observable. The network can have one or more hidden layers. Each neuron in a hidden layer performs a weighted sum of its inputs, passes the sum through an activation function, and sends the result to the neurons in the next layer.
- *Output Layer*: The output layer produces the final predictions or classifications. The number of neurons in this layer depends on the task. For example, in a binary classification problem, there would typically be one neuron that outputs a probability.

2. Neurons and Activation Functions

Each neuron in a neural network performs the following steps:

- *Weighted Sum*: Each input to the neuron is multiplied by a corresponding weight. The neuron computes a weighted sum of its inputs: $z = \sum w_i \times x_i + b$ where w_i are the weights, x_i are the inputs, and b is a bias term.
- *Activation Function*: The weighted sum is passed through an activation function, which introduces non-linearity into the model, allowing the network to learn more complex patterns. Common activation functions include:
    - *Sigmoid*: $\sigma(z) = \frac{1}{1+e^{-z}}$
    - *ReLU (Rectified Linear Unit)*: $ReLU(z) = \max(0, z)$
    - *Tanh (Hyperbolic Tangent):* $Tanh(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$

3. Forward and Backward Propagation

- *Forward Propagation*: During forward propagation, the input data passes through the network, layer by layer, until it reaches the output layer. At each neuron, the weighted sum is calculated, passed through the activation function, and propagated to the next layer.
- *Loss Function*: After forward propagation, the network's prediction is compared to the actual target using a loss function (e.g., Mean Squared Error for regression, Cross-Entropy Loss for classification). The loss function quantifies how well the network is performing.
- *Backward Propagation (Backpropagation):* During backpropagation, the error is propagated back through the network, updating the weights and biases to minimize the loss. This is typically done using an optimization algorithm like Gradient Descent, which adjusts the weights in the direction that reduces the error.

4. Training a Neural Network

- *Epochs and Batches*: Training a neural network involves multiple iterations (epochs) over the dataset. In each epoch, the data is typically divided into smaller batches, and the model updates its weights after each batch (mini-batch gradient descent) or after the entire dataset (batch gradient descent).
- *Optimization Algorithms*: Optimization algorithms like Stochastic Gradient Descent (SGD), Adam, RMSprop, and others are used to adjust the weights and biases of the network. These algorithms determine the learning rate and how the model converges to a solution.

5. Applications of Neural Networks

- *Image Recognition*: Neural networks, particularly Convolutional Neural Networks (CNNs), are widely used in image classification, object detection, and image segmentation.
- *Natural Language Processing (NLP):* Recurrent Neural Networks (RNNs), Long Short-Term Memory (LSTM) networks, and Transformers are commonly used in tasks like text classification, sentiment analysis, and machine translation.
- *Time Series Forecasting*: Neural networks are used to predict future values in time series data, such as stock prices, weather patterns, and sales forecasting.
- *Speech Recognition*: Neural networks are the backbone of modern speech recognition systems, converting spoken language into text.
- *Generative Models*: Networks like Generative Adversarial Networks (GANs) are used to generate new data that resembles the training data, such as creating realistic images or deepfake videos.

6. Pros and Cons of Neural Networks

Pros:

- Flexibility: Can model complex, non-linear relationships.
- Scalability: Can handle large datasets and learn intricate patterns.
- Versatility: Can be applied to a wide range of problems, including classification, regression, and unsupervised learning.

Cons:

- Computationally Intensive: Training deep neural networks can require significant computational resources, including GPUs.
- Data Hungry: Neural networks typically require large amounts of labeled data to perform well.
- Interpretability: Neural networks are often considered "black boxes," making it difficult to interpret how they make decisions.

There are at least a score of different neural network models and variations on each—this is another topic we could literally stretch out for an entire semester—but let’s look at the 10 most popular.

1. Feedforward Neural Network (FNN)

- *Description*: This is the simplest form of a neural network, where information moves in one direction—from the input layer to the output layer—without any cycles or loops. It typically consists of an input layer, one or more hidden layers, and an output layer.
- *Use Cases*: Image and pattern recognition, simple regression and classification tasks.
- *Pros*: Easy to implement and understand, suitable for simple tasks.
- *Cons*: Limited in capturing complex relationships or temporal dependencies.

2. Convolutional Neural Network (CNN)

- *Description*: CNNs are specialized neural networks designed to process data with a grid-like topology, such as images. They consist of convolutional layers that apply filters to the input data, followed by pooling layers that reduce dimensionality, and fully connected layers for classification.
- *Use Cases*: Image classification, object detection, video analysis, and more.
- *Pros*: Excellent at capturing spatial hierarchies in data, like edges, textures, and patterns in images.
- *Cons*: Requires a large amount of labeled data and computational resources.

3. Recurrent Neural Network (RNN)

- *Description*: RNNs are designed to handle sequential data. Unlike feedforward networks, RNNs have loops that allow information to persist, which helps in processing time-series or sequence data. Each neuron can have connections to both the next layer and neurons within the same layer, allowing the network to remember previous inputs.
- *Use Cases*: Time series forecasting, natural language processing, speech recognition.
- *Pros*: Good at handling temporal dependencies and sequential data.
- *Cons*: Can suffer from vanishing or exploding gradient problems, making training difficult for long sequences.

4. Long Short-Term Memory (LSTM) Network

- *Description*: LSTMs are a type of RNN designed to overcome the vanishing gradient problem by incorporating memory cells that can maintain information for longer periods. They have three gates—input, forget, and output gates—that control the flow of information.
- *Use Cases*: Time series forecasting, machine translation, text generation, and other sequence modeling tasks.
- *Pros*: Capable of learning long-term dependencies and remembering patterns over long sequences.
- *Cons*: More complex and computationally expensive than standard RNNs.

5. Gated Recurrent Unit (GRU)

- *Description:* GRUs are a variant of LSTMs that simplify the architecture by combining the forget and input gates into a single update gate. This makes GRUs less complex and faster to train compared to LSTMs, while still handling the vanishing gradient problem effectively.
- *Use Cases*: Similar to LSTMs—time series analysis, text generation, etc.
- *Pros*: Simpler architecture, faster training, and often performs comparably to LSTMs.
- *Cons*: May not be as powerful as LSTMs in some cases, particularly with very long sequences.

6. Autoencoder

- *Description*: Autoencoders are unsupervised neural networks that learn to compress data into a lower-dimensional space (encoding) and then reconstruct the original input from this space (decoding). They consist of an encoder and a decoder part, where the encoder compresses the data and the decoder reconstructs it.
- *Use Cases*: Dimensionality reduction, anomaly detection, data denoising, and generative modeling.
- *Pros*: Useful for feature extraction and dimensionality reduction, can learn complex data representations.

- *Cons*: Can be difficult to train effectively, and overfitting can be a problem if not regularized properly.

7. Generative Adversarial Network (GAN)

- *Description*: GANs consist of two neural networks—a generator and a discriminator—that are trained together. The generator tries to create realistic data (e.g., images) from random noise, while the discriminator tries to distinguish between real and generated data. The two networks compete against each other, improving the generator's output over time.
- *Use Cases*: Image generation, video synthesis, style transfer, and data augmentation.
- *Pros*: Capable of producing highly realistic and complex data, especially in image generation.
- *Cons*: Training is notoriously difficult and unstable, with issues like mode collapse.

8. Radial Basis Function Network (RBFN)

- *Description*: RBFNs are a type of neural network where the hidden layer neurons use a radial basis function (often Gaussian) as their activation function. The output layer performs a weighted sum of these activations, making them useful for function approximation.
- *Use Cases*: Function approximation, time series prediction, and control systems.
- *Pros*: Good for interpolation tasks and can approximate any continuous function given enough neurons.
- *Cons*: Computationally expensive and can suffer from overfitting if not regularized properly.

9. Self-Organizing Map (SOM)

- *Description*: SOMs are a type of unsupervised learning neural network used for clustering and visualization. They consist of a grid of neurons that represent different clusters of the input data. During training, the network organizes itself so that similar input data is grouped together in the grid.
- *Use Cases*: Data visualization, dimensionality reduction, clustering, and anomaly detection.
- *Pros*: Provides a low-dimensional representation of data, making it easier to visualize high-dimensional datasets.
- *Cons*: Can be less effective than other clustering methods and is sensitive to the initial configuration.

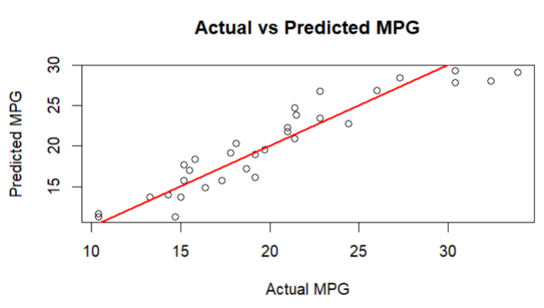
10. Transformer Networks

- *Description*: Transformers are neural networks designed to handle sequence data, particularly in the context of natural language processing. They use self-attention mechanisms to weigh the importance of different words in a sequence, allowing them to capture long-range dependencies without the need for recurrent connections.
- *Use Cases*: Machine translation, text generation, sentiment analysis, and other NLP tasks.
- *Pros*: Highly effective at capturing context in sequences, scalable, and state-of-the-art performance in many NLP tasks.
- *Cons*: Computationally expensive and requires large amounts of data and computing power for training.

Let's look at an implementation of the Feed Forward Network on a regression task.

```
# Load the dataset
data <- as.matrix(mtcars[, -1])  # Use all features except for mpg
target <- mtcars$mpg  # Target is mpg
```

```r
# Normalize the data (standard scaling)
data <- scale(data)
target <- scale(target)

# Initialize Parameters
set.seed(123)
input_size <- ncol(data)
hidden_size <- 10  # Increased hidden size for more complexity
output_size <- 1  # Single output for regression

# Initialize weights and biases
W1 <- matrix(rnorm(input_size * hidden_size, 0, 0.1), nrow = input_size, ncol = hidden_size)
b1 <- matrix(0, nrow = 1, ncol = hidden_size)

W2 <- matrix(rnorm(hidden_size * output_size, 0, 0.1), nrow = hidden_size, ncol = output_size)
b2 <- matrix(0, nrow = 1, ncol = output_size)

# Activation Function: Sigmoid
sigmoid <- function(x) {
  1 / (1 + exp(-x))
}

# Sigmoid Derivative
sigmoid_derivative <- function(x) {
  sigmoid(x) * (1 - sigmoid(x))
}

# Mean Squared Error Loss
mean_squared_error <- function(y_true, y_pred) {
  mean((y_true - y_pred)^2)
}

# Forward Pass
forward_pass <- function(x) {
  z1 <- x %*% W1 + matrix(rep(b1, nrow(x)), nrow = nrow(x), byrow = TRUE)
  a1 <- sigmoid(z1)
  y_pred <- a1 %*% W2 + matrix(rep(b2, nrow(a1)), nrow = nrow(a1), byrow = TRUE)

  return(list(y_pred = y_pred, a1 = a1, z1 = z1))
}

# Backpropagation
backpropagation <- function(x, y, forward_cache, learning_rate = 0.01) {
  y_pred <- forward_cache$y_pred
  a1 <- forward_cache$a1
  z1 <- forward_cache$z1
  # Loss Derivative
  dL_dy_pred <- 2 * (y_pred - y) / nrow(y)

  # Gradients for W2 and b2
  dL_dW2 <- t(a1) %*% dL_dy_pred
  dL_db2 <- colSums(dL_dy_pred)

  # Gradients for W1 and b1
  dL_da1 <- dL_dy_pred %*% t(W2)
  dL_dz1 <- dL_da1 * sigmoid_derivative(z1)

  dL_dW1 <- t(x) %*% dL_dz1
  dL_db1 <- colSums(dL_dz1)

  # Update parameters
  W1 <<- W1 - learning_rate * dL_dW1
  b1 <<- b1 - learning_rate * dL_db1
  W2 <<- W2 - learning_rate * dL_dW2
  b2 <<- b2 - learning_rate * dL_db2
}

# Training the Network
num_epochs <- 5000  # Increase number of epochs
learning_rate <- 0.01  # Adjust the learning rate if needed

for (i in 1:num_epochs) {
  forward_cache <- forward_pass(data)
  y_pred <- forward_cache$y_pred

  if (i %% 500 == 0) {
    loss <- mean_squared_error(target, y_pred)
    cat("Epoch:", i, "Loss:", loss, "\n")
  }

  backpropagation(data, target, forward_cache, learning_rate)
}

# Final Predictions
final_predictions <- forward_pass(data)$y_pred

# Reverse the standard scaling to interpret results
final_predictions <- final_predictions * attr(target, "scaled:scale") + attr(target, "scaled:center")

# Plot the Results
plot(mtcars$mpg, final_predictions, main = "Actual vs Predicted MPG",
     xlab = "Actual MPG", ylab = "Predicted MPG")
abline(0, 1, col = "red", lwd = 2)
```



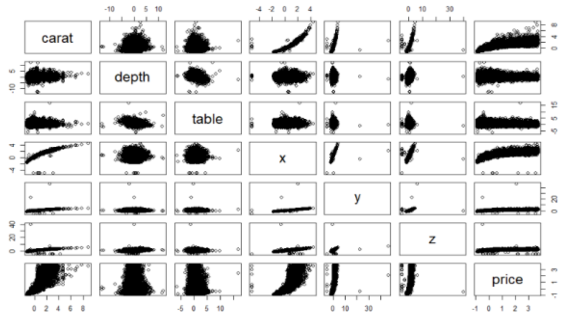
Neural networks are mathematically complex to implement, and in this particular example, the graph makes it look like the predicted values are being treated as labels and not continuous values. One could try changing the activation function to see if that changes the result.

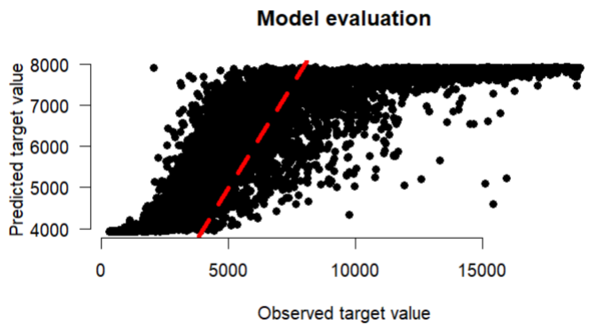
Let's look at another example of a simple neural network.

```
#simple nn
# for manipulating data
library(dplyr)
# for parallel processing
library(purrr)
# for neural network functionality, install first
library(torch)

url_data <- "https://raw.githubusercontent.com/mikeyEcology/Dada-Analysis-on-Diamonds-
Dataset/main/diamonds.csv"

raw_data <- url_data |>
  url() |>
  vroom::vroom()
continuous_features <- c("carat", "depth", "table", "x", "y", "z")
target_name <- "price"

features <- raw_data |>
  dplyr::select(dplyr::all_of(continuous_features)) |>
  scale() |>
  # scale converts the data into an unfortunate format. Make it back to df
  tibble::as_tibble()

target <- raw_data |>
  dplyr::select(dplyr::all_of(target_name)) |>
  scale() |>
  tibble::as_tibble()

train_val <- dplyr::bind_cols(features, target)

apply(train_val, 2, mean)
apply(train_val, 2, sd)

pairs(train_val)

dset <- torch::dataset(
  name = "dset",

  initialize = function(indices) {
    data <- self$prepare_data(train_val[indices, ])
    self$x <- data[[1]]
    self$y <- data[[2]]
  },

  .getitem = function(i) {
    x <- self$x[i, ]
    y <- self$y[i, ]

    list(x, y)
  },

  .length = function() {
    dim(self$y)[1]
  },

  prepare_data = function(input) {
    feature_cols <- input |>
      dplyr::select(dplyr::all_of(continuous_features)) |>
      as.matrix()

    target_col <- input |>
      dplyr::select(dplyr::all_of(target_name)) |>
      as.matrix()

    list(
      torch_tensor(feature_cols),
      torch_tensor(target_col)
    )
  }
)

train_indices <- sample(1:nrow(train_val), size = floor(0.8 * nrow(train_val)))
valid_indices <- setdiff(1:nrow(train_val), train_indices)

# bigger batch sizes train faster. General concensus is that this is not a hyperparameter worth tuning.
batch_size = 256

train_ds <- dset(train_indices)
```

```
train_dl <- train_ds |>
  dataloader(batch_size = batch_size, shuffle = TRUE)

valid_ds <- dset(valid_indices)
valid_dl <- valid_ds |>
  dataloader(batch_size = batch_size, shuffle = FALSE)

net <- nn_module(
  "net",

  initialize = function(num_numerical, fc1_dim
  ) {
    self$fc1 <- nn_linear(num_numerical, fc1_dim)
    self$output <- nn_linear(fc1_dim, 1)
  },

  forward = function(x) {
    x |>
      self$fc1() |>
      nnf_relu() |>
      self$output() |>
      nnf_sigmoid()
  }
)

# This is the number of neurons in the hidden layer of the neural network
fc1_dim <- 32
# The number of features is the number of neurons in the input layer
num_numerical <- length(continuous_features)

# build this neural net
model <- net(num_numerical, fc1_dim)

device <- if (cuda_is_available()) torch_device("cuda:0") else "cpu"

model <- model$to(device = device)

learning_rate <- 0.01
# using stochastic gradient descent for the optimizer
optimizer <- optim_sgd(model$parameters, lr = learning_rate, momentum = 0)
num_epochs <- 10
# using mean squared error loss
loss_func <- nnf_mse_loss

for (epoch in 1:num_epochs) {

  model$train()
  train_losses <- c()
  coro::loop(for (b in train_dl) {
    optimizer$zero_grad()
    output <- model(b[[1]]$to(device = device))
    loss <- loss_func(output, b[[2]]$to(dtype = torch_float(), device = device))
    loss$backward()
    optimizer$step()
    train_losses <- c(train_losses, loss$item())
  })

  model$eval()
  valid_losses <- c()

  coro::loop(for (b in valid_dl) {
    output <- model(b[[1]]$to(device = device))
    loss <- loss_func(output, b[[2]]$to(dtype = torch_float(), device = device))
    valid_losses <- c(valid_losses, loss$item())
  })

  cat(sprintf("Epoch %d: training loss: %3f, validation loss: %3f\n", epoch, mean(train_losses),
mean(valid_losses)))
}

# we set model to evaluation model so it knows not to calculate gradients now
model$eval()

# predict the target for the validation dataset
target_pred <- c()
coro::loop(for (b in valid_dl) {
  output <- model(b[[1]]$to(device = device))
  #target_pred <- c(target_pred, output)
  for (i in 1:length(output)) {
    pred_array <- as_array(output)[i,]
    target_pred <- c(target_pred, pred_array)
  }
})

# get the observed target from the validation dataset
# we use `as_array` to convert from tensors to standard R data structures
target_obs <- as_array(valid_ds$y)

# get mean and std dev from original data (these were used when scaling by the `scale` function)
raw_target <- raw_data |>
  dplyr::select(dplyr::all_of(target_name)) |>
  dplyr::pull()
std_dev <- sd(raw_target)
mn <- mean(raw_target)
```

```
# unscale
y_obs <- (target_obs * std_dev) + mn
y_pred <- (target_pred *std_dev) + mn

plot(
  y_obs, y_pred, main = "Model evaluation",
  xlab = "Observed target value",
  ylab = "Predicted target value",
  axes = FALSE, pch = 16
)
segments(
  x0=0, y0=0, x1=max(y_obs), y1=max(y_obs), lty = 2, lwd = 4, col = "red"
)
axis(1)
axis(2, las = 2)

# Pearson correlation
cor(y_obs, y_pred, method = "pearson")
# Spearman correlation
cor(y_obs, y_pred, method = "spearman")
```



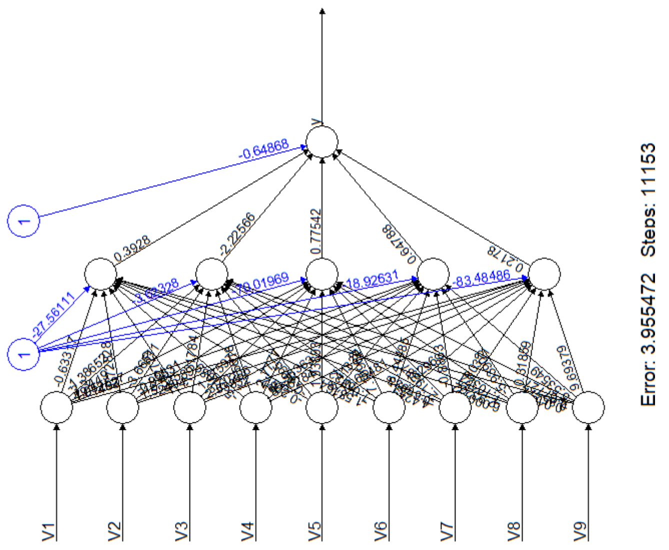


Neural networks from scratch are extremely advanced coding tasks, even the simplest of them.

Let’s look at a simple implementation using a package. Many of the more complex neural network implementations are built on packages that use Python and you would need to install Python and get it to talk to R in order to run them. For our examples, we’ll use packages that don’t depend on Python.

```
library(mlbench)
data("BreastCancer")

#Clean off rows with missing data
BreastCancer = BreastCancer[which(complete.cases(BreastCancer)==TRUE),]

head(BreastCancer)

names(BreastCancer)

y = as.matrix(BreastCancer[,11])
```

```
y[which(y=="benign")] = 0
y[which(y=="malignant")] = 1
y = as.numeric(y)
x = as.numeric(as.matrix(BreastCancer[,2:10]))
x = matrix(as.numeric(x),ncol=9)

library(deepnet)
nn <- nn.train(x, y, hidden = c(5))
yy = nn.predict(nn, x)
print(head(yy))

yhat = matrix(0,length(yy),1)
yhat[which(yy > mean(yy))] = 1
yhat[which(yy <= mean(yy))] = 0
cm = table(y,yhat)
print(cm)

print(sum(diag(cm))/sum(cm))

library(neuralnet)
df = data.frame(cbind(x,y))
nn = neuralnet(y~V1+V2+V3+V4+V5+V6+V7+V8+V9,data=df,hidden = 5)
yy = nn$net.result[[1]]
yhat = matrix(0,length(y),1)
yhat[which(yy > mean(yy))] = 1
yhat[which(yy <= mean(yy))] = 0
print(table(y,yhat))

plot(nn)
```



Resources:

1. https://www.geeksforgeeks.org/building-a-simple-neural-network-in-r-programming/#
2. https://www.datacamp.com/tutorial/neural-network-models-r
3. https://rviews.rstudio.com/2020/07/20/shallow-neural-net-from-scratch-using-r-part-1/
4. https://rpubs.com/mikeyt/simple_nn_in_r
5. https://www.r-bloggers.com/2021/04/deep-neural-network-in-r/
6. https://www.projectpro.io/recipes/make-simple-neural-network-r
7. https://tensorflow.rstudio.com/examples/image_classification_from_scratch
8. https://srdas.github.io/DLBook/DeepLearningWithR.html