Neural Networks

Naeemullah Khan
naeemullah.khan@kaust.edu.sa

KAUST Academy
King Abdullah University of Science and Technology

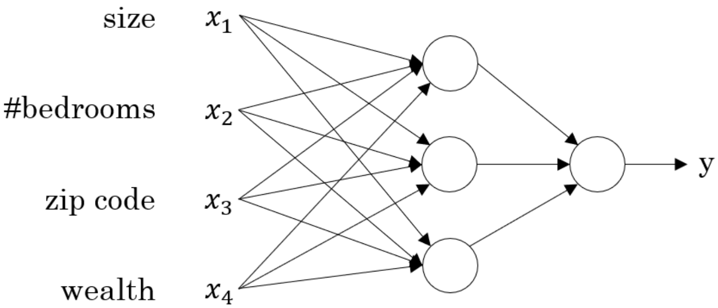
November 29, 2025

Table of Content

- A neural network is a function that maps input features to outputs using layers of simple units called neurons
- Each neuron computes a weighted sum of its inputs then applies a non linear activation
- Stacking layers lets the model learn complex patterns that linear models cannot capture

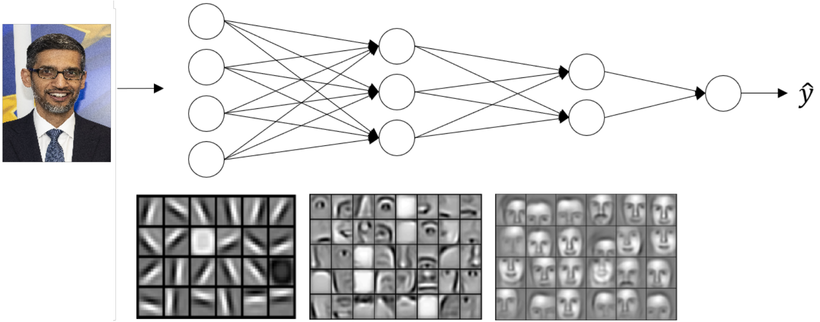
- **Input layer** holds the features
- **Hidden layers** learn intermediate representations
- **Output layer** produces predictions

Neural Network



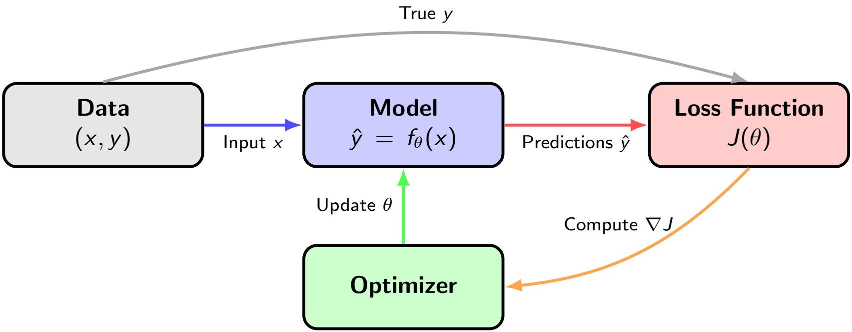
Why Multiple Layers? (Depth)

- A single hidden layer network is already very powerful in theory
- In practice, **deeper** networks:
  - Construct more powerful intermediate features
  - Can represent complex functions more **efficiently** (fewer neurons overall)



Neural Network as Part of the Learning System

- **Neural network model**
  - Maps input x to prediction $\hat{y} = f_\theta(x)$
  - θ are the parameters (weights and biases)
- **Loss function**
  - Measures how far predictions are from targets
  - Guides what it means to be a good model for the task
- **Optimizer**
  - Uses gradients of the loss to update θ
  - Tries to find parameters that reduce the loss on data



Forward Pass:

- Data $\rightarrow$ Model $\rightarrow$ Predictions
- Compute Loss

Backward Pass:

- Calculate gradients
- Update parameters

Repeat until loss converges (model learns)!

The Forward Pass

- The forward pass starts from inputs and flows through the network
- Each layer applies a linear transformation then an activation
- At the end we compute the loss between predictions and targets

Example for one hidden layer

$$a^{(1)} = \sigma\left(W^{(1)}x + b^{(1)}\right)$$

$$\hat{y} = g\left(W^{(2)}a^{(1)} + b^{(2)}\right)$$

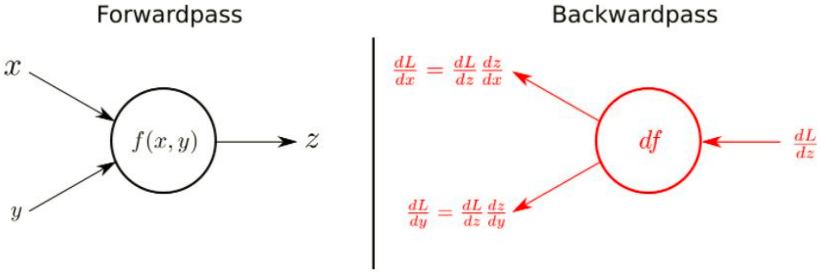
- $x \in \mathbb{R}^k$ input features
- $a^{(1)}$ hidden layer activations
- g is the output activation chosen for the task

The Backward Pass

- The backward pass starts from the loss and flows backward through the network
- It uses the chain rule to compute gradients of the loss with respect to every parameter
- These gradients tell the optimizer how to change weights to reduce the loss

At a high level

- Forward pass gives $\hat{y}$ and loss $J(\theta)$
- Backward pass gives gradients $\nabla_\theta J$
- Optimizer updates parameters using these gradients


Forwardpass
x
y
$f(x, y)$
z
Backwardpass
$\frac{dL}{dx} = \frac{dL}{dz}\frac{dz}{dx}$
df
$\frac{dL}{dz}$
$\frac{dL}{dy} = \frac{dL}{dz}\frac{dz}{dy}$

Single Neuron Computation

- Input features $x \in \mathbb{R}^k$
- Weights $w \in \mathbb{R}^k$ and bias $b \in \mathbb{R}$

$$z = w^\top x + b \quad , \quad a = \sigma(z)$$

- z is the pre activation (linear part)
- σ is an activation function
- a is the neuron output that feeds the next layer or the loss

Why We Need Activation Functions

- If we only stack linear layers without activations we still get a linear function
- Non linear activations let the network approximate complex functions
- They introduce bends and thresholds that help separate classes and model non linear trends

ReLU

$$\sigma(z) = \max(0, z)$$

- Simple and works well in deep nets
- Keeps positive values and drops negatives

Sigmoid

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

- Outputs values in $(0, 1)$
- Used for probabilities in binary classification

Tanh

$$\sigma(z) = \tanh(z)$$

- Outputs values in $(-1, 1)$
- Often used in recurrent networks

Softmax for Multiclass Outputs

- For multiclass classification the output layer gives scores $z \in \mathbb{R}^C$ for C classes
- The softmax function turns scores into a probability distribution

$$\text{softmax}(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{C} e^{z_j}}$$

- Each output is between 0 and 1
- All outputs sum to 1

Inputs and Outputs in Practice

- Input vector $x \in \mathbb{R}^k$
  - Example features
  - Age income number of past purchases
- Output shape depends on the task
  - Scalar for single output regression or binary classification
  - Vector for multi output regression or multiclass classification

Network Shapes for Different Tasks

Task	Input	Output of NN	Typical loss
Single output regression	$x \in \mathbb{R}^k$	$\hat{y} \in \mathbb{R}$	Mean squared error
Multi output regression	$x \in \mathbb{R}^k$	$\hat{y} \in \mathbb{R}^m$	MSE over all outputs
Binary classification	$x \in \mathbb{R}^k$	$\hat{p} \in (0, 1)$	Binary cross entropy
Multiclass classification	$x \in \mathbb{R}^k$	$\hat{p} \in \mathbb{R}^C$ softmax	Cross entropy

Examples of Heads on Top of the Same Body

- The hidden layers can be shared across tasks
- Only the last layer and the loss change

- Regression head
  - Last layer has one neuron with linear activation
  - Use mean squared error
- Binary head
  - Last layer has one neuron with sigmoid
  - Use binary cross entropy
- Multiclass head
  - Last layer has C neurons with softmax
  - Use cross entropy over classes

Optimizers: Gradient Descent

- Parameters θ are updated in the direction that reduces the loss

$$\theta_{\text{new}} = \theta_{\text{old}} - \alpha \nabla_{\theta} J(\theta)$$

- α is the learning rate
- $\nabla_{\theta} J(\theta)$ is the gradient of the loss with respect to the parameters
- In deep learning we usually use mini batch gradient descent

AdaGrad

- Keeps a running sum of squared gradients for each parameter:

$$G_t = G_{t-1} + g_t^2$$

- Update rule with parameter wise learning rate:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t} + \epsilon} g_t$$

- Parameters with large accumulated gradients get smaller effective learning rates
- Works well for sparse features

RMSProp

- Uses an exponential moving average of squared gradients:

$$E[g^2]_t = \alpha E[g^2]_{t-1} + (1-\alpha) g_t^2$$

- Update rule:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

- Prevents the learning rate from shrinking too fast and keeps training going
- Often used as a base idea in modern optimizers

Adam

- Keeps a moving average of gradients and squared gradients:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

- Bias correction:

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \qquad \hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

- Update rule:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t}+\epsilon}\,\hat{m}_t$$

- Combines momentum + adaptive step size and usually speeds up training

Optimizers: AdamW

AdamW

- Uses the same Adam update, plus decoupled weight decay:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \; - \; \eta\lambda\theta_t$$

- Weight decay is applied directly to parameters instead of through the gradient
- Often used as a strong default optimizer for modern deep networks

Key Training Hyperparameters

- **Learning rate** η
  - Too small: training is very slow
  - Too large: loss may bounce or diverge
- **Batch size**
  - Number of examples per parameter update
  - Small batch: noisy but often good generalization
  - Large batch: smoother gradients, needs more memory
- **Epochs**
  - One pass over the whole training dataset (consists of batches/steps)

What We Choose When Building a Neural Network

- **Model side**
  - Number of layers and neurons per layer
  - Activation functions (`ReLU`, `sigmoid`, `tanh`, . . .)
  - Output layer shape and activation
- **Training side**
  - Loss function (MSE, cross entropy, . . .)
  - Optimizer (SGD, Adam, AdamW, . . .)
  - Hyperparameters (learning rate, batch size, epochs)

Underfitting and Overfitting in Neural Networks

- **Underfitting**:
  - Model is too simple or not trained enough
- **Overfitting**:
  - Model memorizes training data

How Can We Detect Overfitting?

- Idea: measure performance on **data the model did not see during training**
- We split our dataset:
  - A part for **training** the model
  - A part for **validation/testing** to check generalization
- If the model keeps improving on the training set but stops improving (or gets worse) on the validation set, it is overfitting.

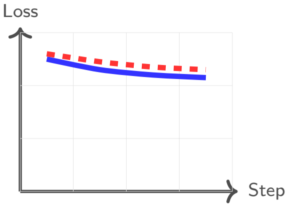
Hold Out Split: Train, Validation, Test

- We split the dataset into **disjoint** parts
  - **Train set** to fit the model
  - **Validation set** to tune models and hyperparameters
  - **Test set** kept aside for a final unbiased estimate
- For simplicity, people usually just split into train and test.
- We usually **shuffle** the data rows before splitting
  - So that all splits follow the same distribution as the full data
  - To avoid easy leakage such as first half of customers in train and last half in test

Training vs Validation Loss

Underfit

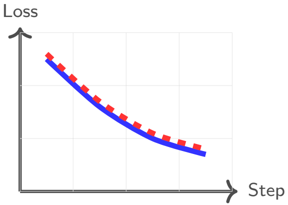
Model too simple



Train MSE: 2.2
Val MSE: 2.3

Good Fit

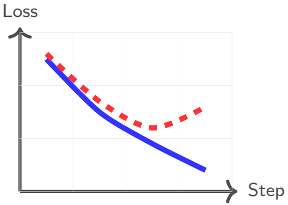
Optimal balance



Train MSE: 0.6
Val MSE: 0.7

Overfit

Memorizing data



Train MSE: 0.3
Val MSE: 1.4

Train loss   Validation loss

Reading Loss Curves

- During training we usually plot:
  - Training loss vs epochs
  - Validation loss vs epochs

These curves are one of the most useful debugging tools in deep learning.

How To Fix Underfitting and Overfitting

If the model is underfitting

- Use a more complex model
- Train longer
- Increase the number of layers and neurons

If the model is overfitting

- Simplify the model
- Get more data
- Add regularization

Regularization: Controlling Model Complexity

- **Goal:** Keep the model powerful but prevent overfitting

L2 Regularization (Weight Decay)

- Add a penalty on large weights to the loss:

$$J_{\text{total}}(\theta) = J(\theta) + \lambda \|\theta\|_2^2$$

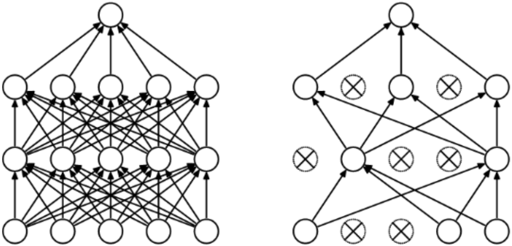
- Encourages smaller, smoother weights that generalize better

Early Stopping

- Monitor validation loss during training
- Stop training when validation loss stops improving
- Simple and very effective in practice

Regularization: Dropout

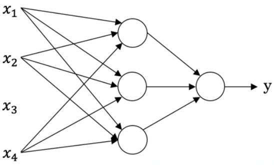
- During training, **randomly turn off** (drop) some neuron outputs
- Each mini-batch sees a slightly different subnet of the network
- Intuition: the network cannot rely on a single neuron and learns more robust features
- At test time, we use the **full** network but scale activations appropriately



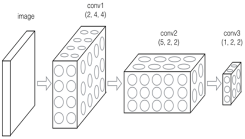
(a) Standard Neural Net (b) After applying dropout.

Types of Neural Network Architectures

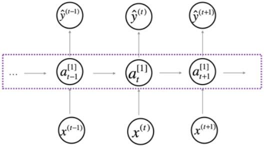
- **MLP** fully connected network for tabular data
- **CNN** convolutional neural network for images
- **RNN and variants** for sequences and time series



Standard NN



Convolutional NN

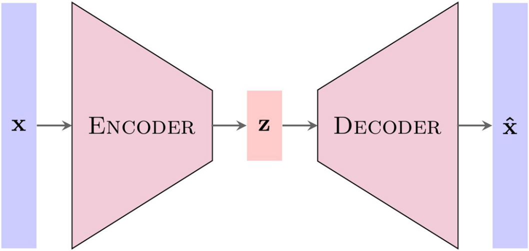


Recurrent NN

What Is an Autoencoder

- An autoencoder is a network that tries to reconstruct its input
- It has two parts
  - **Encoder** maps x to a lower dimensional code z
  - **Decoder** maps z back to a reconstruction $\hat{x}$
- The loss compares $\hat{x}$ and x usually with mean squared error

Autoencoder Diagram



The bottleneck forces the network to learn a compressed representation of the data

Autoencoder Applications

- Dimensionality reduction and visualization
- Denoising images or signals
- Anomaly detection using high reconstruction error
- As a pretraining step to learn useful representations

Autoencoder Example: Anomaly Detection

- Train an autoencoder to reconstruct **normal** examples (e.g., normal transactions)
- At test time:
  - Feed an input x through the autoencoder to get $\hat{x}$
  - Compute reconstruction error: $||x - \hat{x}||$
- If error is **small**: input looks similar to training data (likely normal)
- If error is **large**: input is unusual (possible anomaly)
- Very useful when we have many normal samples and few labeled anomalies

References

- Andrew Ng, Deep Learning Specialization and Machine Learning course
- Aurélien Géron, Hands On Machine Learning with Scikit Learn Keras and TensorFlow
- Ian Goodfellow Yoshua Bengio Aaron Courville, Deep Learning
- DeepLearning.AI short courses on optimization and advanced architectures