

# CS-470: Machine Learning

Week 10 - TensorFlow and Keras for Deep Neural Networks

Instructor: Dr. Sajjad Hussain

Department of Electrical and Computer Engineering  
SEECS, NUST

November 20th, 2025

# Overview

- 1 Introduction to TensorFlow and Keras
- 2 Building Neural Networks
  - Sequential API
  - Functional API
  - Model Subclassing
- 3 Understanding Layers and Activations
- 4 Model Compilation
- 5 Training Models
- 6 Enhancing Training with Callbacks
- 7 Model Evaluation and Deployment
- 8 Best Practices and Summary

# What is TensorFlow?

## TensorFlow

An end-to-end open-source platform for machine learning developed by Google

- **Production Ready:** Scalable for production environments
- **Flexibility:** From research to production with same framework
- **Keras Integration:** High-level API for fast prototyping
- **Cross-Platform:** Run on CPU, GPU, TPU, mobile, and web

## Keras

High-level neural networks API, now the official high-level API for TensorFlow

# Why TensorFlow/Keras?

## Advantages:

- Easy to learn and use
- Excellent documentation
- Large community
- Production deployment tools
- Pre-trained models
- Visualization tools  
(TensorBoard)

## Perfect for:

- Rapid prototyping
- Research and development
- Production systems
- Educational purposes
- Transfer learning

# Installation and Setup

```
1 # Install TensorFlow
2 pip install tensorflow
3
4 # For GPU support
5 pip install tensorflow-gpu
6
7 # Basic import
8 import tensorflow as tf
9 from tensorflow import keras
10
11 # Check version and GPU
12 print(f"TensorFlow version: {tf.__version__}")
13 print(f"GPU Available: {tf.config.
    list_physical_devices('GPU')}")
```

## Note

Always check compatibility between TensorFlow, CUDA, and cuDNN

# Three Ways to Build Models

- ① **Sequential API:** Linear stack of layers (Simplest)
- ② **Functional API:** Complex architectures with multiple inputs/outputs
- ③ **Model Subclassing:** Maximum flexibility with custom classes (Advanced)

## Recommendation for Beginners

Start with Sequential API, progress to Functional API for complex models

# Sequential API: Concept

## Idea

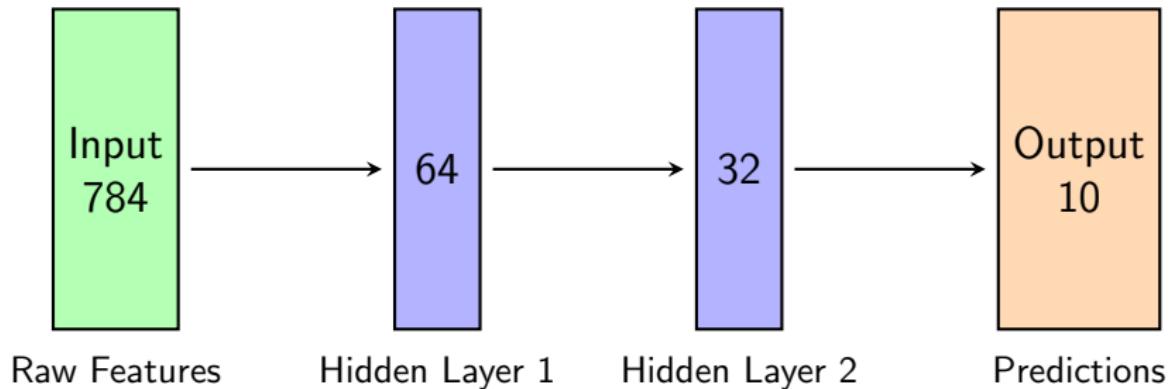
The Sequential API allows you to create models layer-by-layer in a linear stack

- **Simple and intuitive:** Perfect for beginners
- **Limited to single input, single output:** Cannot create complex architectures
- **Two creation methods:** List of layers or adding layers sequentially
- **Most common for feedforward networks:** CNNs, simple classifiers

## Limitation

Cannot create models with Multiple inputs, Multiple outputs, Shared layers, and Residual connections.

# Neural Network: Left-to-Right Information Flow



## How Information Flows

- **Forward Pass:** Input data propagates through each layer sequentially
- **Feature Transformation:** Each layer learns to extract increasingly abstract features
- **Final Output:** Network produces classification probabilities

# Sequential API: Implementation

```
1 from tensorflow.keras import Sequential
2 from tensorflow.keras.layers import Dense
3
4 # Method 1: List of layers (Recommended)
5 model = Sequential([
6     Dense(64, activation='relu', input_shape=(784,)),
7     Dense(32, activation='relu'),
8     Dense(10, activation='softmax')
9 ])
10
11 # Method 2: Add layers sequentially
12 model = Sequential()
13 model.add(Dense(64, activation='relu', input_shape=(784,)))
14 model.add(Dense(32, activation='relu'))
15 model.add(Dense(10, activation='softmax'))
```

# Functional API: Concept

## Idea

The Functional API creates models by connecting layers explicitly, allowing complex architectures

- **More flexible:** Can create any architecture
- **Explicit connections:** You define how layers connect
- **Multiple inputs/outputs:** Supports complex model designs
- **Shared layers:** Layers can be reused multiple times

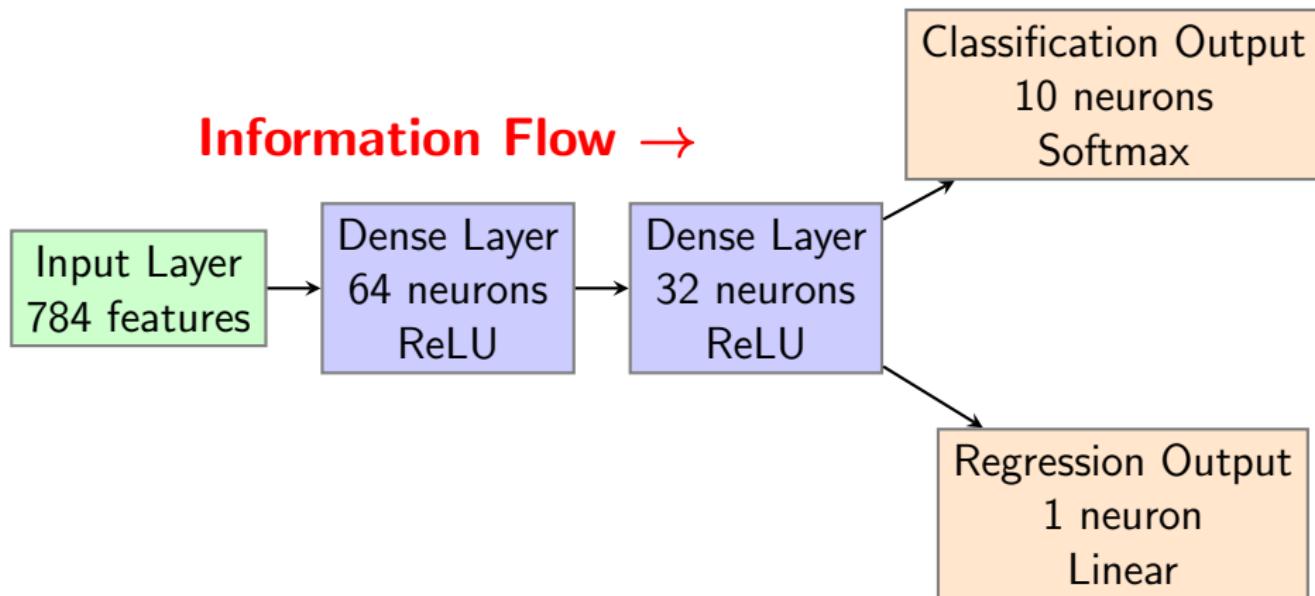
## When to Use Functional API

- Multi-input models
- Multi-output models
- Models with residual connections
- Any non-sequential architecture

# Functional API: Basic Implementation

```
1 from tensorflow.keras import Model
2 from tensorflow.keras.layers import Input, Dense
3
4 # Define input tensor
5 inputs = Input(shape=(784,))
6
7 # Create layers by calling them on tensors
8 x = Dense(64, activation='relu')(inputs)
9 x = Dense(32, activation='relu')(x)
10
11 # Define output
12 outputs = Dense(10, activation='softmax')(x)
13
14 # Create model by specifying inputs and outputs
15 model = Model(inputs=inputs, outputs=outputs)
16
17 # View model architecture
18 model.summary()
```

# Functional API: Multi-Output Network Architecture



# Corresponding Code using Functional API

```
1 # Single input
2 inputs = Input(shape=(784,))
3
4 # Shared layers
5 x = Dense(64, activation='relu')(inputs)
6 x = Dense(32, activation='relu')(x)
7
8 # Multiple outputs
9 classification_output = Dense(10, activation='softmax')(x)
10 regression_output = Dense(1, activation='linear')(x)
11
12 # Create multi-output model
13 model = Model(inputs=inputs,
14 outputs=[classification_output, regression_output])
```

# Model Subclassing: Concept

## Idea

Create models by subclassing the Model class and defining the forward pass in the call method

- **Maximum flexibility:** Complete control over forward pass
- **Pythonic approach:** Uses object-oriented programming
- **Complex logic:** Can include any Python control flow
- **Research-oriented:** Ideal for experimental architectures

## Advanced Technique

- Requires deeper understanding of OOP and TensorFlow
- More error-prone for beginners
- Model architecture is less explicit
- **Recommended for advanced users only**

# Model Subclassing: Basic Example

```
1 class SimpleClassifier(tf.keras.Model):
2     def __init__(self):
3         super(SimpleClassifier, self).__init__()
4         # Define layers in __init__
5         self.dense1 = Dense(64, activation='relu')
6         self.dropout = Dropout(0.2)
7         self.dense2 = Dense(32, activation='relu')
8         self.classifier = Dense(10, activation='softmax')
9
10    def call(self, inputs, training=False):
11        # Define forward pass
12        x = self.dense1(inputs)
13        x = self.dropout(x, training=training)
14        x = self.dense2(x)
15        return self.classifier(x)
16
17 model = SimpleClassifier() # Create model instance
```



# Common Layer Types

Layer Type	Purpose	Common Use
Dense	Fully connected	All neural networks, final classification layers
Dropout	Regularization	Prevent overfitting by randomly dropping units
BatchNormalization	Normalization	Improve training stability and speed
Conv2D	2D convolution	Image processing, feature extraction
MaxPooling2D	Downsampling	Reduce spatial dimensions in CNNs
LSTM/GRU	Recurrent layers	Sequence data, time series, text
Embedding	Word embeddings	NLP tasks, categorical data

# Activation Functions: Purpose

## What are Activation Functions?

Mathematical functions that determine the output of a neural network node given its input

- **Introduce non-linearity:** Without them, neural networks would be linear models
- **Determine output range:** Different functions have different output ranges
- **Affect learning:** Choice impacts gradient flow and training stability
- **Problem-specific:** Different functions suit different problem types

Activation functions are what allow neural networks to learn complex, non-linear relationships in data.

# Common Activation Functions

Function	Range	Common Use
ReLU	$[0, \infty)$	Hidden layers (most common)
Sigmoid	$(0, 1)$	Binary classification output
Tanh	$(-1, 1)$	Hidden layers (zero-centered)
Softmax	$(0, 1)$	Multi-class classification output
Linear	$(-\infty, \infty)$	Regression output

## Practical Recommendations

- Use **ReLU** for hidden layers (fast training)
- Use **Sigmoid** for binary classification output
- Use **Softmax** for multi-class classification output
- Use **Linear** for regression output

# Using Activation Functions in Practice

```
1 from tensorflow.keras.layers import Dense  
2  
3 #Method-1 As string in layer definition (Most common)  
4 model.add(Dense(64, activation='relu'))  
5  
6 # Method 2: As a separate layer  
7 from tensorflow.keras.layers import Activation  
8 model.add(Dense(64))  
9 model.add(Activation('relu'))  
10  
11 # Method 3: Using activation function directly  
12 from tensorflow.keras.activations import relu  
13  
14 previous_layer = ...  
15 x = Dense(64)(previous_layer)  
16 x = relu(x)
```

# Using Activation Functions in Practice

```
1 # Binary Classification
2 model = Sequential([
3     Dense(64, activation='relu'),      # Hidden layer
4     Dense(32, activation='relu'),      # Hidden layer
5     Dense(1, activation='sigmoid'))   # Output layer
6
7 # Multiclass classification
8 model = Sequential([
9     Dense(64, activation='relu'),      # Hidden layer
10    Dense(32, activation='relu'),      # Hidden layer
11    Dense(1, activation='sigmoid'))   # Output layer
12
13 # Regression
14 model = Sequential([
15     Dense(64, activation='relu'),      # Hidden layer
16     Dense(32, activation='relu'),      # Hidden layer
17     Dense(1, activation='linear'))    # Output layer
```



# Model Compilation: Purpose

## What is Model Compilation?

The process of configuring the model for training by specifying:

- **Optimizer:** How the model updates its weights
- **Loss function:** How the model measures its performance
- **Metrics:** How to monitor training progress

## Important

You must compile a model before training it. This step configures the learning process.

# Loss Functions: Choosing the Right One

---

<b>Problem Type</b>	<b>Loss Function</b>
Binary Classification	<code>binary_crossentropy</code>
Multi-class Classification	<code>categorical_crossentropy</code>
Multi-class (integer labels)	<code>sparse_categorical_crossentropy</code>
Regression	<code>mean_squared_error</code>
Robust Regression	<code>mean_absolute_error</code>

---

# Optimizers: How Models Learn

Optimizers are the algorithms that update model weights to minimize the loss function during training.

Optimizer	Common Use	Description
Adam	Default choice	Adaptive learning rate, works well for most problems
SGD	Fine-tuning	With momentum, good for transfer learning
RMSprop	RNNs	Good for recurrent networks

## Practical Advice

- Start with **Adam** - it works well for most problems
- Use **SGD with momentum** for fine-tuning pre-trained models
- Adjust learning rate based on problem complexity

# Compilation in Practice

```
1 # For classification problems
2 model.compile(optimizer='adam',
3 loss='categorical_crossentropy',
4 metrics=['accuracy', 'precision', 'recall'])
5
6 # For regression problems
7 model.compile(optimizer='rmsprop', loss='mse', metrics
8     =['mae'])
9
10 # With custom optimizer parameters
11 from tensorflow.keras.optimizers import Adam
12
13 model.compile(optimizer=Adam(learning_rate=0.001),
14 loss='categorical_crossentropy',
15 metrics=['accuracy', 'precision', 'recall'])
```

# Training Process: Overview

- **Forward pass:** Input data flows through the network to produce predictions
- **Loss calculation:** Compare predictions with true values using loss function
- **Backward pass:** Calculate gradients (how to change weights to reduce loss)
- **Weight update:** Optimizer adjusts weights based on gradients
- **Repeat:** Process continues for multiple epochs

## Key Concepts

- **Epoch:** One complete pass through the entire training dataset
- **Batch:** A subset of the training data processed together
- **Iteration:** Number of batches needed to complete one epoch

# Basic Training Setup

```
1 # Simple training with numpy arrays
2 history = model.fit(
3     x_train, y_train,      # Training data
4     batch_size=32,         # Samples per gradient update
5     epochs=100,            # Number of training cycles
6     validation_data=(x_val, y_val),  # Data for validation
7     verbose=1              # Progress display
8 )
9
10 # Using TensorFlow Dataset Class (For large data)
11 train_dataset = tf.data.Dataset.from_tensor_slices((
12     x_train, y_train))
13 train_dataset = train_dataset.batch(32).shuffle(1000)
14
15 history = model.fit(train_dataset, epochs=100,
16     validation_data=val_dataset)
```

# Understanding Training Output

## What to Look For During Training

- **Training loss:** Should decrease over time
- **Validation loss:** Should also decrease (watch for divergence)
- **Metrics:** Accuracy, precision, etc. should improve
- **Overfitting:** When validation loss stops improving or increases

## Healthy Training Pattern

- Both training and validation loss decrease
- Training and validation metrics improve
- Loss curves smooth out over time
- No large gap between training and validation performance

# Callbacks: Automated Training Assistants

## What are Callbacks?

Objects that can perform actions at various stages of training to enhance and automate the process

- **Automate common tasks:** Save models, adjust learning rates, stop early
- **Monitor training:** Track metrics, detect issues
- **Improve results:** Better models with less manual intervention
- **Save time:** Automate repetitive tasks

## Common Use Cases

- Save the best model automatically
- Stop training when model stops improving
- Reduce learning rate when progress stalls
- Visualize training with TensorBoard

# Essential Callbacks for Every Project

```
1 # 1. Early stopping: Stop when validation loss stops
  improving
2 early_stop = tf.keras.callbacks.EarlyStopping(
3     monitor='val_loss',      # Metric to monitor
4     patience=10,            # Epochs to wait before stopping
5     restore_best_weights=True) # Keep best model weights
6
7 # 2. Reduce learning rate when progress stalls
8 reduce_lr = tf.keras.callbacks.ReduceLROnPlateau(
9     monitor='val_loss',
10    factor=0.5,             # Reduce LR by half
11    patience=5,              # Wait 5 epochs
12    min_lr=1e-7)            # Minimum learning rate
```

# Essential Callbacks for Every Project

```
1
2 # 3. Save best model automatically
3 check_point = tf.keras.callbacks.ModelCheckpoint(
4   'best_model.h5',
5   monitor='val_accuracy',
6   save_best_only=True)
7
8 # Create list of callbacks
9 callbacks = [early_stop, reduce_lr, check_point]
10
11 # Use callbacks in training
12 history = model.fit(x_train, y_train, epochs=100,
13 callbacks=callbacks, validation_data=(x_val, y_val))
```

# Model Evaluation: Beyond Accuracy

## Comprehensive Evaluation Strategy

Good model evaluation involves multiple metrics and visualization techniques

- **Quantitative metrics:** Numbers that measure performance
- **Qualitative analysis:** Visual inspection of results
- **Error analysis:** Understanding where model fails
- **Comparison:** Against baseline models and human performance

## Evaluation Checklist

- Calculate multiple relevant metrics
- Analyze confusion matrix for classification
- Plot learning curves
- Compare with simple baselines

# Comprehensive Model Evaluation

```
1 # Detailed metrics for classification
2 from sklearn.metrics import classification_report,
3     confusion_matrix
4
5 # Basic evaluation
6 test_loss, test_accuracy = model.evaluate(x_test,
7     y_test, verbose=0)
8 print(f"Test accuracy: {test_accuracy:.4f}")
9
10 # Generate predictions
11 predictions = model.predict(x_test)
12 predicted_classes = np.argmax(predictions, axis=1)
13
14 print("1. Classification Report:")
15 print(classification_report(y_test, predicted_classes))
```

# Comprehensive Model Evaluation

```
1 print("2. Confusion Matrix:")
2 print(confusion_matrix(y_test, predicted_classes))
3
4 # For regression problems
5 from sklearn.metrics import mean_squared_error,
6     r2_score
7
8 mse = mean_squared_error(y_test, predictions)
9 r2 = r2_score(y_test, predictions)
10 print(f"MSE: {mse:.4f}, R^2: {r2:.4f}")
```

# Model Saving and Deployment

## Saving Models for Future Use

TensorFlow provides multiple ways to save and load models

- **Complete model:** Architecture + weights + optimizer state
- **Weights only:** Just the learned parameters
- **Architecture only:** Just the model structure
- **TensorFlow SavedModel:** Standard format for deployment

## When to Use Each Method

- **Complete model:** Resume training later
- **Weights only:** Share trained models
- **Architecture only:** Reuse model design
- **SavedModel:** Production deployment

# Saving and Loading Models

```
1 # Save entire model (recommended)
2 model.save('my_model.h5')    # HDF5 format
3
4 # Save only weights
5 model.save_weights('model_weights.h5')
6
7 # Save only architecture
8 with open('model_architecture.json', 'w') as f:
9     f.write(model.to_json())
```

# Saving and Loading Models

```
1 # Load complete model
2 loaded_model = tf.keras.models.load_model('my_model.h5')
3
4 # Load weights into existing architecture
5 model.load_weights('model_weights.h5')
6
7 # Load architecture and then weights
8 with open('model_architecture.json', 'r') as f:
9     model = tf.keras.models.model_from_json(f.read())
10    model.load_weights('model_weights.h5')
11
12 # Make predictions with loaded model
13 predictions = loaded_model.predict(new_data)
```

# Model Design Best Practices

## Architecture Design Principles

- **Start simple:** Begin with a basic model and gradually add complexity
- **Use appropriate layers:** Choose layers that match your data type
- **Include regularization:** Use Dropout and BatchNorm to prevent overfitting
- **Monitor capacity:** Balance model complexity with dataset size
- **Use pre-trained models:** Leverage transfer learning when possible

# Model Design Best Practices

## Training Strategy

- **Use validation data:** Always monitor performance on unseen data
- **Implement early stopping:** Prevent overfitting automatically
- **Use learning rate scheduling:** Adapt learning rate during training
- **Monitor multiple metrics:** Get complete picture of performance
- **Use data augmentation:** Increase effective dataset size

# Common Issues and Solutions

## Common Problems:

- Model not learning
- Overfitting
- Training too slow
- Memory issues
- Unstable training

## Possible Solutions:

- Check learning rate
- Add regularization
- Use smaller batches
- Optimize data pipeline
- Add BatchNorm

## Debugging Strategy

- ① Start with a simple baseline
- ② Verify data loading and preprocessing
- ③ Check model can overfit small dataset
- ④ Monitor training curves carefully
- ⑤ Experiment systematically

# Summary: TensorFlow/Keras Workflow

## Standard Deep Learning Pipeline

- ① **Data Preparation:** Load, clean, and preprocess data
- ② **Model Design:** Choose architecture using Sequential or Functional API
- ③ **Model Compilation:** Specify optimizer, loss, and metrics
- ④ **Model Training:** Fit model to data with callbacks
- ⑤ **Model Evaluation:** Assess performance on test data
- ⑥ **Model Deployment:** Save and use model for predictions

## Key Success Factors

- **Good data:** Quality and quantity matter most
- **Appropriate architecture:** Match model to problem type
- **Proper training:** Use validation and callbacks