Artificial Neural Networks and TensorFlow - Comprehensive Exam Notes

1. TensorFlow Overview

What is TensorFlow?

- **Definition**: Python-friendly open source library for numerical computation
- **Purpose**: Well-suited for large-scale machine learning and deep learning
- **Key Features**:
  - Define computation graphs in Python
  - Breaks graphs into chunks for parallel execution
  - Supports multiple CPU, GPU, and TPU processing

TensorFlow with Keras

- Keras is the high-level API for TensorFlow
- Provides user-friendly interface for building neural networks
- Simplifies model creation, training, and evaluation

---

2. Linear Threshold Unit (LTU)

Basic Structure

- **Inputs**: Numerical values ($x_1, x_2, ..., x_n$)
- **Weights**: Each input has an associated weight ($w_1, w_2, ..., w_n$)
- **Computation**:
  - Weighted sum: $z = w_1x_1 + w_2x_2 + ... + w_nx_n = w^T x$
  - Output: $\hat{y} = step(z) = step(w^T x)$
- **Step Function**: $step(z) = 0$ if $z < 0$, otherwise 1

Mathematical Representation

- Vector form: $z = w^T x$
- Decision boundary created by step function
- Forms basis for more complex neural networks

---

3. Perceptron

Architecture

- **Structure**: Single layer of neurons
- **Bias**: Added as fixed input value of 1
- **Neuron j computation**: $z_j = w_{j1}x_1 + ... + w_{jn}x_n + b = w_j$^T x + b

Activation Functions (replacing step function)

1. **ReLU (Rectified Linear Unit)**: $ReLU(z_j) = max(0, z_j)$
2. **Sigmoid (Logistic)**: $\sigma(z_j) = e$^$z_j/(1 + e$^$z_j) = 1/(1 + e$^$(-z_j))$
3. **Hyperbolic Tangent**: $tanh(z_j) = 2\sigma(2z_j) - 1$

Training Process

- **Weight Update**: $w_{i,j}(new) = w_{i,j} - \alpha \cdot \partial J(w_j,b_j)/\partial w_{i,j}$
- **Bias Update**: $b_j(new) = b_j - \alpha \cdot \partial J(w_j,b_j)/\partial b_j$
- **Learning Rate**: α (step size parameter)

---

4. Multi-Layer Perceptron (MLP)

Architecture Components

- **Input Layer**: Conceptual layer that forwards inputs
- **Hidden Layer(s)**: One or more layers between input and output
- **Output Layer**: Produces final predictions
- **Connections**: Fully connected layers (except output layer includes bias)

Key Properties

- **Universal Approximation**: Sufficiently large MLP can approximate any continuous function
- **Feedforward**: Information flows in one direction (input → hidden → output)
- **Bias Neurons**: Special neurons that always output 1

Hidden Layer Computation

- **Input**: $h = (h_1, ..., h_n)$ from previous layer
- **Computation**: $z_j = w_j$^T h + b
- **Activation Functions**: Sigmoid, Tanh, or ReLU

5. Output Layer Configurations

For Regression

- **Linear Output**: $\hat{y} = w_j$^T h + b_j (no activation)
- **Bounded Output**: $\hat{y} = \sigma(w_j$^T h + $b_j)$ (sigmoid activation)
- **Positive Output**: $\hat{y} = \mathrm{ReLU}(w_j$^T h + $b_j)$

For Classification

- **Softmax Function**:
  - Intermediate: $z_j = w_j$^T h + b_j
  - Output: $\hat{y}_j$ = softmax_j(z) = e^z_j / $\Sigma_{i=1}^{n}$ e^z_i
  - Interpretation: $\hat{y}_j$ represents probability of class j

---

6. Cost Functions

Regression

- **Mean Squared Error (MSE)**: Standard loss function for regression problems
- **Formula**: $J = (1/m) \Sigma_{i=1}^{m} (y_i - \hat{y}_i)^2$

Classification

- **Cross-Entropy Loss**:
  - Single sample: cross_entropy(y_i, $\hat{y}_i$) = $-\Sigma_j y_{j,i} \log(\hat{y}_{j,i})$
  - Total cost: cost(y, ŷ) = $(1/m) \Sigma_{i=1}^{m}$ cross_entropy(y_i, $\hat{y}_i$)
- **Binary Classification**: Binary cross-entropy with sigmoid activation

---

7. Training Process (Backpropagation)

Forward Pass

1. **Input Processing**: Feed training instance x to network
2. **Layer-by-layer Computation**: Calculate outputs for each layer
3. **Prediction**: Compute final output $\hat{y} = f(x)$
4. **Error Calculation**: Measure cost(y, ŷ)

Backward Pass

1. **Output Layer Error**: Calculate error contribution from each output neuron
2. **Hidden Layer Error**: Work backwards to measure error contribution from each hidden neuron
3. **Gradient Calculation**: Compute gradients for all weights and biases
4. **Weight Update**: Apply gradient descent to reduce error

Key Concepts

- **Chain Rule**: Used to compute gradients through multiple layers
- **Gradient Descent**: Iterative optimization algorithm
- **Learning Rate**: Controls step size in weight updates

---

8. Keras Implementation

Dataset Preparation

python

```python
# Example: Fashion MNIST
fashion_mnist = keras.datasets.fashion_mnist
(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist.load_data()

# Scaling and splitting
X_valid, X_train = X_train_full[:5000] / 255.0, X_train_full[5000:] / 255.0
y_valid, y_train = y_train_full[:5000], y_train_full[5000:]
```

Model Architecture (Sequential API)

python

```python
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),  # 2D to 1D conversion
    keras.layers.Dense(300, activation="relu"),   # Hidden layer 1
    keras.layers.Dense(100, activation="relu"),   # Hidden layer 2
    keras.layers.Dense(10, activation="softmax")  # Output layer
])
```

Model Compilation

python

```python
model.compile(
    loss="sparse_categorical_crossentropy",  # For sparse labels
    optimizer="sgd",                          # Stochastic Gradient Descent
    metrics=["accuracy"]                      # Performance metric
)
```

Training

python

```python
history = model.fit(
    X_train, y_train,
    epochs=30,
    validation_data=(X_valid, y_valid)
)
```

Evaluation and Prediction

python

```python
# Evaluate model
model.evaluate(X_test, y_test)

# Make predictions
y_proba = model.predict(X_new)
y_pred = np.argmax(y_proba, axis=-1)
```

---

9. Hyperparameter Configurations

Regression Networks

Parameter	Typical Values
Input neurons	One per feature
Hidden layers	1-20 layers
Neurons per layer	10x-100x input features
Output neurons	1 per target variable
Hidden activation	ReLU, Sigmoid, Tanh
Output activation	None, ReLU, Sigmoid/Tanh
Loss function	MSE

Classification Networks

Task	Binary	Multi-label	Multi-class
Output neurons	1	1 per label	1 per class
Output activation	Sigmoid	Sigmoid	Softmax
Loss function	Binary Cross-entropy	Binary Cross-entropy	Categorical Cross-entropy

---

10. Advanced Techniques

Regularization

- **L1 Regularization**: $\tilde{J}(W) = J(W) + \lambda_1 \Sigma_i|w_i|$
- **L2 Regularization**: $\tilde{J}(W) = J(W) + \lambda_2 \Sigma_i w_i^2$
- **Combined**: $\tilde{J}(W) = J(W) + \lambda_1 \Sigma_i|w_i| + \lambda_2 \Sigma_i w_i^2$

Implementation in Keras

```python
keras.layers.Dense(
    4,
    activation="relu",
    kernel_regularizer=keras.regularizers.l1_l2(l1=0.01, l2=0.01)
)
```

Early Stopping

- **Purpose**: Prevent overfitting
- **Method**: Stop training when validation error stops decreasing

- **Implementation**: Monitor validation loss during training

---

11. Hyperparameter Tuning

Grid Search with Keras Tuner

```python
def build_model(hp):
    n_hidden = hp.Int("n_hidden", min_value=0, max_value=8, default=2)
    n_neurons = hp.Int("n_neurons", min_value=16, max_value=256)
    learning_rate = hp.Float("learning_rate", min_value=1e-4, max_value=1e-2)

    model = tf.keras.Sequential()
    model.add(tf.keras.layers.Flatten())
    for _ in range(n_hidden):
        model.add(tf.keras.layers.Dense(n_neurons, activation="relu"))
    model.add(tf.keras.layers.Dense(10, activation="softmax"))

    model.compile(
        loss="sparse_categorical_crossentropy",
        optimizer=tf.keras.optimizers.SGD(learning_rate=learning_rate),
        metrics=["accuracy"]
    )
    return model
```

---

12. Loss Functions in Keras

Common Loss Functions

- **sparse_categorical_crossentropy**: For integer labels (0, 1, 2, ...)
- **categorical_crossentropy**: For one-hot encoded labels
- **binary_crossentropy**: For binary classification
- **mse**: Mean Squared Error for regression

Selection Criteria

- **Sparse labels**: Use sparse_categorical_crossentropy
- **One-hot labels**: Use categorical_crossentropy
- **Binary tasks**: Use binary_crossentropy
- **Regression**: Use mse or other regression losses

13. Optimizers

Available Optimizers in Keras

- **SGD**: Standard Stochastic Gradient Descent
- **Adam**: Adaptive Moment Estimation
- **RMSprop**: Root Mean Square Propagation
- **Adagrad**: Adaptive Gradient Algorithm
- **Adadelta**: Extension of Adagrad

Choosing Optimizers

- **SGD**: Simple, interpretable, good baseline
- **Adam**: Generally good performance, adaptive learning rates
- **RMSprop**: Good for recurrent neural networks

---

14. Activation Functions

Available Functions

- **ReLU**: Most common for hidden layers
- **Sigmoid**: Output layer for binary classification
- **Softmax**: Output layer for multi-class classification
- **Tanh**: Alternative to sigmoid
- **Linear**: No activation (regression output)

Selection Guidelines

- **Hidden layers**: Usually ReLU
- **Binary output**: Sigmoid
- **Multi-class output**: Softmax
- **Regression output**: Linear or ReLU (for positive values)

---

15. Initialization Strategies

Available Initializers

- **GlorotNormal/GlorotUniform**: Good for sigmoid/tanh

- **HeNormal**: Good for ReLU
- **Constant**: Initialize to specific value
- **Random**: Various random initializations

Usage

```python
keras.layers.Dense(10, activation="relu", kernel_initializer="he_normal")
```

---

16. Key Exam Tips

Important Concepts to Remember

1. **Architecture Design**: Know when to use different layer types and sizes
2. **Activation Functions**: Understand when to use each type
3. **Loss Functions**: Match loss function to problem type
4. **Regularization**: Understand L1/L2 regularization effects
5. **Training Process**: Understand forward/backward pass
6. **Hyperparameter Tuning**: Know common ranges and selection criteria

Common Exam Questions

1. **Design networks**: Given a problem, specify architecture
2. **Choose parameters**: Select appropriate loss, optimizer, activation
3. **Debug training**: Identify overfitting/underfitting issues
4. **Mathematical understanding**: Compute forward/backward pass
5. **Implementation**: Write Keras code for specific architectures

Problem-Solving Approach

1. **Identify problem type**: Regression vs. classification
2. **Design architecture**: Input → Hidden → Output layers
3. **Choose components**: Activation, loss, optimizer
4. **Consider regularization**: If overfitting is likely
5. **Plan evaluation**: Metrics and validation strategy