Artificial Neural Networks Lecture Notes

Notes Prepared By: Chandan Chaudhari
GitHub: https://github.com/chandanc5525

Contents

1 Introduction to Artificial Neural Networks

Artificial Neural Networks are computational models inspired by the biological nervous system. The fundamental processing unit, the neuron, mimics its biological counterpart through the following components:

- **Dendrites**: Input receivers (feature vectors)
- **Cell Body**: Processing unit (activation function)
- **Axon**: Output transmitter (prediction)
- **Synapses**: Adaptive connections (weights)

ANNs learn mappings from input space $\mathcal{X}$ to output space $\mathcal{Y}$ through parameterized function approximation:

$$f : \mathcal{X} \to \mathcal{Y}, \quad f(\mathbf{x}) = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b}) \tag{1}$$

where $\mathbf{W}$ represents weight matrices, $\mathbf{b}$ denotes bias vectors, and σ represents non-linear activation functions.

2 ANN Architecture and Mathematics

2.1 Fundamental Components

A standard feedforward neural network comprises:

- **Input Layer**: $\mathbf{a}^{(0)} = \mathbf{x} \in \mathbb{R}^{n_0}$
- **Hidden Layers**: $\mathbf{a}^{(l)} = \sigma(\mathbf{z}^{(l)})$, where $\mathbf{z}^{(l)} = \mathbf{W}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}$
- **Output Layer**: $\hat{\mathbf{y}} = \mathbf{a}^{(L)}$

For layer l with n_l neurons:
Pre-activation:

$$\mathbf{z}^{(l)} = \mathbf{W}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)} \quad \text{where } \mathbf{W}^{(l)} \in \mathbb{R}^{n_l \times n_{l-1}}, \mathbf{b}^{(l)} \in \mathbb{R}^{n_l} \tag{2}$$

Activation:

$$\mathbf{a}^{(l)} = \sigma\left(\mathbf{z}^{(l)}\right) \tag{3}$$

Final Output:

$$\hat{\mathbf{y}} = \mathbf{a}^{(L)} = f(\mathbf{x}; \theta) \tag{4}$$

where $\theta = \{\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \ldots, \mathbf{W}^{(L)}, \mathbf{b}^{(L)}\}$ represents all learnable parameters.

2.2 Backpropagation Mathematics

The backpropagation algorithm efficiently computes gradients using the chain rule:

Output Layer Gradient:

$$\delta^{(L)} = \nabla_{\mathbf{a}^{(L)}} \mathcal{L} \odot \sigma'(\mathbf{z}^{(L)}) \tag{5}$$

Hidden Layer Gradients:

$$\delta^{(l)} = \left((\mathbf{W}^{(l+1)})^\top \delta^{(l+1)}\right) \odot \sigma'(\mathbf{z}^{(l)}) \tag{6}$$

Parameter Gradients:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(l)}} = \delta^{(l)} (\mathbf{a}^{(l-1)})^\top \tag{7}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(l)}} = \delta^{(l)} \tag{8}$$

Using gradient descent with learning rate α:

$$\mathbf{W}^{(l)} \leftarrow \mathbf{W}^{(l)} - \alpha \frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(l)}} \tag{9}$$

$$\mathbf{b}^{(l)} \leftarrow \mathbf{b}^{(l)} - \alpha \frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(l)}} \tag{10}$$

3 Activation Functions

3.1 Comprehensive Mathematical Analysis

3.1.1 Sigmoid Function

Definition:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{11}$$

Derivative:

$$\sigma'(x) = \sigma(x)(1 - \sigma(x)) \tag{12}$$

3.1.2 Hyperbolic Tangent (tanh)

Definition:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = 2\sigma(2x) - 1 \tag{13}$$

Derivative:

$$\tanh'(x) = 1 - \tanh^2(x) \tag{14}$$

3.1.3 Rectified Linear Unit (ReLU)

Definition:

$$\text{ReLU}(x) = \max(0, x) \tag{15}$$

Derivative:

$$\text{ReLU}'(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases} \tag{16}$$

3.1.4 Leaky ReLU

Definition:

$$\text{LeakyReLU}(x) = \max(\alpha x, x), \quad \alpha \in (0, 1) \tag{17}$$

Derivative:

$$\text{LeakyReLU}'(x) = \begin{cases} 1 & \text{if } x > 0 \\ \alpha & \text{if } x \leq 0 \end{cases} \tag{18}$$

3.1.5 Softmax Function

Definition (for multi-class classification):

$$\text{softmax}(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{19}$$

3.2 Activation Function Selection Guide

Scenario	Recommended Function	Mathematical Justification
Hidden Layers	ReLU/Leaky ReLU	Computational efficiency, avoids vanishing grad
Binary Classification Output	Sigmoid	Outputs interpretable as probabilities
Multi-class Output	Softmax	Ensures probability distribution
Regression Output	Linear/Identity	Unbounded output range
RNN Hidden Layers	Tanh	Zero-centered, handles negative values

Table 1: Activation Function Selection Guidelines

4 Batch Normalization: Theory and Implementation

4.1 Mathematical Formulation

Given a mini-batch $\mathcal{B} = \{x_1, \ldots, x_m\}$:

Batch Statistics:

$$\mu_{\mathcal{B}} = \frac{1}{m} \sum_{i=1}^{m} x_i \tag{20}$$

$$\sigma_{\mathcal{B}}^2 = \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \tag{21}$$

Normalization:

$$\hat{x}_i = \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \tag{22}$$

Scale and Shift:

$$y_i = \gamma \hat{x}_i + \beta \tag{23}$$

where γ (scale) and β (shift) are learnable parameters.

During backpropagation, gradients flow through the normalization transformation:

$$\frac{\partial \mathcal{L}}{\partial x_i} = \frac{\partial \mathcal{L}}{\partial y_i} \cdot \gamma \cdot \left(\frac{1}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} - \frac{(x_i - \mu_{\mathcal{B}})^2}{m(\sigma_{\mathcal{B}}^2 + \epsilon)^{3/2}} \right) \tag{24}$$

4.2 Theoretical Significance and Benefits

Internal Covariate Shift Reduction:

- Stabilizes distribution of layer inputs during training
- Allows higher learning rates without divergence
- Restores training stability in deep networks

Regularization Effect:

- Adds noise through mini-batch statistics
- Reduces overfitting without explicit dropout
- Improves generalization performance

4.3 Practical Implementation

```python
import tensorflow as tf
from tensorflow.keras.layers import BatchNormalization

model = tf.keras.Sequential([
    tf.keras.layers.Dense(128, input_shape=(784,)),
    BatchNormalization(),
    tf.keras.layers.Activation('relu'),
    tf.keras.layers.Dropout(0.3),

    tf.keras.layers.Dense(64),
    BatchNormalization(),
    tf.keras.layers.Activation('relu'),
    tf.keras.layers.Dropout(0.3),

    tf.keras.layers.Dense(10, activation='softmax')
])

inputs = tf.keras.layers.Input(shape=(784,))
x = tf.keras.layers.Dense(128)(inputs)
x = BatchNormalization()(x)
x = tf.keras.layers.Activation('relu')(x)
x = tf.keras.layers.Dropout(0.3)(x)

x = tf.keras.layers.Dense(64)(x)
x = BatchNormalization()(x)
x = tf.keras.layers.Activation('relu')(x)
x = tf.keras.layers.Dropout(0.3)(x)

outputs = tf.keras.layers.Dense(10, activation='softmax')(x)
model = tf.keras.Model(inputs=inputs, outputs=outputs)
```

Listing 1: Batch Normalization Implementation

4.4 Batch Normalization Best Practices

Aspect	Recommendation	Reason
Placement	After Dense/Conv, before Activation	Normalizes inputs to activation function
Training vs Inference	Use different modes	Training uses batch stats, inference uses mov
Batch Size	Use larger batches (> 32)	More stable statistics estimation
Learning Rate	Can increase learning rate	BN stabilizes training dynamics
Initialization	Less sensitive to initialization	BN reduces dependence on initial weights

Table 2: Batch Normalization Best Practices

5 Optimization Algorithms

5.1 Stochastic Gradient Descent (SGD)

Basic SGD:

$$\theta_{t+1} = \theta_t - \alpha \nabla_\theta \mathcal{L}(\theta_t) \tag{25}$$

SGD with Momentum:

$$v_{t+1} = \beta v_t + (1 - \beta) \nabla_\theta \mathcal{L}(\theta_t) \tag{26}$$

$$\theta_{t+1} = \theta_t - \alpha v_{t+1} \tag{27}$$

5.2 Adaptive Optimization Methods

5.2.1 RMSprop

Mean Square Update:

$$E[g^2]_t = \rho E[g^2]_{t-1} + (1 - \rho) g_t^2 \tag{28}$$

Parameter Update:

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{E[g^2]_t + \epsilon}} g_t \tag{29}$$

5.2.2 Adam (Adaptive Moment Estimation)

First Moment:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{30}$$

Second Moment:

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \tag{31}$$

Bias Correction:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{32}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{33}$$

Parameter Update:

$$\theta_{t+1} = \theta_t - \frac{\alpha \hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{34}$$

5.3 Optimizer Selection Matrix

Optimizer	Use Cases	Advantages	Hyperparameters
SGD	Convex problems, fine control	Theoretical guarantees, simple	Learning rate, momentum
Adam	Most deep learning tasks	Fast convergence, adaptive	$\alpha, \beta_1, \beta_2, \epsilon$
RMSprop	RNNs, non-stationary objectives	Good for online learning	α, ρ, ϵ
Adagrad	Sparse data, NLP	Per-parameter learning rates	α, ϵ

Table 3: Optimizer Characteristics and Applications

6 Loss Functions

6.1 Regression Loss Functions

6.1.1 Mean Squared Error (MSE)

Definition:

$$\mathcal{L}_{\text{MSE}} = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \tag{35}$$

Gradient:

$$\frac{\partial \mathcal{L}_{\text{MSE}}}{\partial \hat{y}_i} = \frac{2}{n} (\hat{y}_i - y_i) \tag{36}$$

6.1.2 Mean Absolute Error (MAE)

Definition:

$$\mathcal{L}_{\text{MAE}} = \frac{1}{n} \sum_{i=1}^{n} |y_i - \hat{y}_i| \tag{37}$$

6.1.3 Huber Loss

Definition:

$$\mathcal{L}_{\text{Huber}} = \begin{cases} \frac{1}{2}(y - \hat{y})^2 & \text{if } |y - \hat{y}| \leq \delta \\ \delta |y - \hat{y}| - \frac{1}{2}\delta^2 & \text{otherwise} \end{cases} \tag{38}$$

6.2 Classification Loss Functions

6.2.1 Binary Cross-Entropy

Definition:

$$\mathcal{L}_{\text{BCE}} = -\frac{1}{n} \sum_{i=1}^{n} [y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)] \tag{39}$$

Gradient:

$$\frac{\partial \mathcal{L}_{\text{BCE}}}{\partial \hat{y}_i} = \frac{\hat{y}_i - y_i}{\hat{y}_i(1 - \hat{y}_i)} \tag{40}$$

6.2.2 Categorical Cross-Entropy

Definition:

$$\mathcal{L}_{\text{CCE}} = -\frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{K} y_{i,j}\log(\hat{y}_{i,j}) \tag{41}$$

6.3 Loss Function Selection Guide

Problem Type	Recommended Loss	Mathematical Properties
Binary Classification	Binary Cross-Entropy	Maximum likelihood, convex
Multi-class Classification	Categorical Cross-Entropy	Information-theoretic optimal
Regression (Normal errors)	Mean Squared Error	Maximum likelihood for Gaussian
Regression (Robust)	Huber Loss	Combines MSE and MAE benefits
Imbalanced Classification	Focal Loss	Addresses class imbalance

Table 4: Loss Function Selection Guidelines

7 Evaluation Metrics

7.1 Classification Metrics

Accuracy:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{42}$$

Precision:

$$\text{Precision} = \frac{TP}{TP + FP} \tag{43}$$

Recall:

$$\text{Recall} = \frac{TP}{TP + FN} \tag{44}$$

F1-Score:

$$\text{F1} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{45}$$

7.2 Regression Metrics

Mean Absolute Error:

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n} |y_i - \hat{y}_i| \tag{46}$$

Mean Squared Error:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{47}$$

R-squared:

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2} \tag{48}$$

8 Practical Implementation Examples

8.1 Complete ANN Implementation with Batch Normalization

```python
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, BatchNormalization, Dropout
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau
from tensorflow.keras.regularizers import l2
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_classification

class AdvancedANN:
    def __init__(self, input_dim, num_classes=1):
        self.input_dim = input_dim
        self.num_classes = num_classes
        self.model = self._build_advanced_model()

    def _build_advanced_model(self):
        model = Sequential([
            Dense(128, input_shape=(self.input_dim,),
                  kernel_initializer='he_normal',
                  kernel_regularizer=l2(0.001)),
            BatchNormalization(),
            tf.keras.layers.Activation('relu'),
            Dropout(0.4),

            Dense(64, kernel_initializer='he_normal',
                  kernel_regularizer=l2(0.001)),
            BatchNormalization(),
            tf.keras.layers.Activation('relu'),
            Dropout(0.3),

            Dense(32, kernel_initializer='he_normal'),
            BatchNormalization(),
            tf.keras.layers.Activation('relu'),
            Dropout(0.2),

            Dense(self.num_classes,
                  activation='sigmoid' if self.num_classes == 1 else 'softmax',
                  kernel_initializer='glorot_uniform')
        ])
        return model

    def compile_model(self, learning_rate=0.001):
        if self.num_classes == 1:
            loss = 'binary_crossentropy'
            metrics = ['accuracy', 'precision', 'recall', 'auc']
        else:
            loss = 'categorical_crossentropy'
            metrics = ['accuracy', 'categorical_accuracy']

        self.model.compile(
```

```
            optimizer=Adam(learning_rate=learning_rate),
            loss=loss,
            metrics=metrics
        )

    def train(self, X_train, y_train, validation_data=None,
              epochs=100, batch_size=32):
        callbacks = [
            EarlyStopping(
                monitor='val_loss' if validation_data else 'loss',
                patience=15,
                restore_best_weights=True,
                verbose=1
            ),
            ReduceLROnPlateau(
                monitor='val_loss' if validation_data else 'loss',
                factor=0.5,
                patience=8,
                min_lr=1e-7,
                verbose=1
            )
        ]

        history = self.model.fit(
            X_train, y_train,
            batch_size=batch_size,
            epochs=epochs,
            validation_data=validation_data,
            callbacks=callbacks,
            verbose=1,
            shuffle=True
        )
        return history

def demonstrate_ann():
    X, y = make_classification(n_samples=1000, n_features=20,
                               n_redundant=2, n_informative=15,
                               random_state=42)

    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2, random_state=42
    )

    scaler = StandardScaler()
    X_train = scaler.fit_transform(X_train)
    X_test = scaler.transform(X_test)

    ann = AdvancedANN(input_dim=20, num_classes=1)
    ann.compile_model(learning_rate=0.001)

    history = ann.train(
        X_train, y_train,
        validation_data=(X_test, y_test),
        epochs=100,
        batch_size=32
    )

    test_results = ann.model.evaluate(X_test, y_test, verbose=0)
```

```python
    print(f"Test Loss: {test_results[0]:.4f}")
    print(f"Test Accuracy: {test_results[1]:.4f}")

    return ann, history

if __name__ == "__main__":
    model, training_history = demonstrate_ann()
```

Listing 2: Complete ANN Implementation

9 Advanced Topics and Best Practices

9.1 Weight Initialization Strategies

Xavier/Glorot Initialization:

$$W \sim \mathcal{U}\left(-\sqrt{\frac{6}{n_{in}+n_{out}}}, \sqrt{\frac{6}{n_{in}+n_{out}}}\right) \tag{49}$$

He Initialization:

$$W \sim \mathcal{N}\left(0, \sqrt{\frac{2}{n_{in}}}\right) \tag{50}$$

9.2 Regularization Techniques

L2 (Ridge) Regularization:

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{data}} + \lambda \sum_i w_i^2 \tag{51}$$

L1 (Lasso) Regularization:

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{data}} + \lambda \sum_i |w_i| \tag{52}$$

Dropout:

$$a_i^{\text{dropout}} = \begin{cases} \frac{a_i}{p} & \text{with probability } p \\ 0 & \text{with probability } 1-p \end{cases} \tag{53}$$

9.3 Systematic Hyperparameter Optimization

Algorithm 1 Systematic Hyperparameter Optimization for ANNs

Training data D_{train}, Validation data D_{val} Optimal hyperparameters θ^* Define search space: Learning rate: $\alpha \in [10^{-5}, 10^{-1}]$ Architecture: hidden layers, units per layer Batch size: $b \in \{32, 64, 128, 256\}$ Regularization: $\lambda \in [10^{-6}, 10^{-2}]$ Initialize with random search (50 trials) each configuration θ_i Train model M_i with θ_i on D_{train} Evaluate M_i on D_{val} to get performance P_i Select top-k configurations based on P_i Refine with Bayesian optimization around top configurations Validate final model on test set with statistical testing θ^* with confidence intervals

9.4 ANN Architecture Design Principles

Principle	Description	Implementation
Progressive Compression	Gradually reduce layer sizes	$256 \rightarrow 128 \rightarrow 64 \rightarrow 32$
Batch Normalization	Normalize layer inputs	BN after each dense layer
Dropout Regularization	Prevent overfitting	Increasing dropout: $0.1 \rightarrow 0.3 \rightarrow 0.5$
Residual Connections	Improve gradient flow	Skip connections in deep networks
Proper Initialization	Set appropriate starting weights	He/Xavier initialization

Table 5: ANN Architecture Design Principles

10 Conclusion

This comprehensive reference has covered the mathematical foundations, architectural considerations, and practical implementations of Artificial Neural Networks. Key takeaways include:

1. Mathematical Understanding: Deep knowledge of forward/backward propagation enables better architecture design and debugging

2. Batch Normalization: Critical for training deep networks, improves stability and convergence

3. Appropriate Component Selection: Choice of activation functions, optimizers, and loss functions should align with problem characteristics

4. Regularization: Proper use of batch normalization, dropout, and weight regularization prevents overfitting

5. Systematic Evaluation: Comprehensive metrics and analysis ensure robust model performance

References

1. Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep Learning. MIT Press.

2. Bishop, C. M. (2006). Pattern Recognition and Machine Learning. Springer.

3. LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. Nature, 521(7553), 436-444.

4. Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.

5. Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. arXiv preprint arXiv:1502.03167.

A Mathematical Notation Summary

Symbol	Description
$\mathbf{W}^{(l)}$	Weight matrix for layer l
$\mathbf{b}^{(l)}$	Bias vector for layer l
$\mathbf{z}^{(l)}$	Pre-activation values for layer l
$\mathbf{a}^{(l)}$	Activation values for layer l
$\sigma(\cdot)$	Activation function
$\mathcal{L}$	Loss function
α	Learning rate
$\delta^{(l)}$	Error term for layer l
∇	Gradient operator
$\odot$	Element-wise multiplication
$\mathcal{B}$	Mini-batch
$\mu_{\mathcal{B}}$	Batch mean
$\sigma^2_{\mathcal{B}}$	Batch variance
γ, β	Batch normalization parameters

Table 6: Mathematical Notation Summary

B Common Activation Functions and Derivatives

Function	Definition	Derivative
Sigmoid	$\sigma(x) = \frac{1}{1+e^{-x}}$	$\sigma(x)(1 - \sigma(x))$
Tanh	$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	$1 - \tanh^2(x)$
ReLU	$\max(0, x)$	$\begin{cases} 1 & x > 0 \\ 0 & x \leq 0 \end{cases}$
Leaky ReLU	$\max(\alpha x, x)$	$\begin{cases} 1 & x > 0 \\ \alpha & x \leq 0 \end{cases}$
Softmax	$\frac{e^{z_i}}{\sum_j e^{z_j}}$	$\text{softmax}(z_i)(\delta_{ij} - \text{softmax}(z_j))$

Table 7: Activation Functions and Their Derivatives