**1. Постановка задачи**

Цель: построить модель, классифицирующую изображения персонажей как принадлежащих к одной из двух категорий:

* Doom
* Animal Crossing

Пусть имеется обучающая выборка из изображений:

Требуется найти функцию

где θ — параметр нейросети,минимизирующие функцию потерь.

В рамках обучения использовался оптимизатор adam.

Функция потерь, бинарная кросэнтропия:

В качестве функции активации использовалась сигмойда:

**2. Тренировочные и тестовые наборы данных**

|  |  |
| --- | --- |
| **Характеристика** | **Значение** |
| Количество классов | 2 |
| Названия классов | doom, animal\_crossing |
| Общее количество примеров | 1597 изображений |
| Размер обучающих данных | 70% |
| Размер валидационных данных | 20 |
| Размер тестовых данных | 10% |
| Преобразование размеров | 64×64 |

**3. Метрика качества**

Для бинарной классификации используем **(Accuracy)**, а также дополнительно **Precision** и **Recall** для более подробного

где

* (True Positive) — число «doom»-изображений, правильно отнесённых к классу «doom».
* (True Negative) — число «animal\_crossing»-изображений, правильно отнесённых к классу «animal\_crossing».
* (False Positive)— «animal\_crossing», ошибочно помеченные как «doom».
* (False Negative) — «doom», ошибочно помеченные как «animal\_crossing».

**4. Исходный формат хранения данных**

В каталоге данные хранятся в виде:

doom-crossing/

├── animal\_crossing/

│ ├── image1.jpg

│ ├── ...

├── doom/

│ ├── image1.jpg

│ ├── ...

Формат — обычные изображения (.jpg) и (.png) в папках по классам.

**5. Формат подачи данных в сеть**

Входные изображения разбиваются на батчи, затем нормализуются в значения от 0 до 1.

Батчи делятся 70% на обучения 20% на валидацию данных(во время обучения) и 10% на тестирование получившейся модели.

**6. Разработанные программы/скрипты. Листинг кода. Описание разработанных функций.**

Загрузка данных

data = tf.keras.utils.image\_dataset\_from\_directory('/content/doom-crossing', image\_size=(IMG\_WIDTH, IMG\_HEIGHT), batch\_size=BATCH\_SIZE)

# Нормализируем данные

data = data.map(lambda x,y: (x/255, y))

Разбиваем данные на тестовую обучающую и валидационные выборки. Т.к. изображения не разбиваются на батчи идеально, формируется дополнительный неполный батч.

train\_size\_percentage = 0.7

val\_size\_percentage = 0.2

test\_size\_percentage = 0.1

spared\_batches = len(data) % 10

train\_size = int(len(data) \* train\_size\_percentage)

val\_size = int(len(data) \* val\_size\_percentage)

test\_size = int(len(data) \* test\_size\_percentage) + spared\_batches

train = data.take(train\_size)

val = data.skip(train\_size).take(val\_size)

test = data.skip(train\_size + val\_size).take(test\_size)

Задаем архитектуру сети простой полносвязной сети. Слои Rescaling и Flatten нужны для предобработки данных для подачи в сеть. Для обучения используется оптимизатор adam, для функции потерь используется категориальная кросэнтропия, которая применяется для задачи бинарной классификации.

simple\_model = Sequential([

Rescaling(1./255, input\_shape=IMG\_SHAPE),

Flatten(),

Dense(128, activation='relu'),

Dense(1, activation='sigmoid')

])

Функция обучения и тестирования модели. Обучает модель, а затем выводит получившиеся графики потерь и точности, вычисляет точность на тестовых данных.

def train \_model(model, train, val, initial\_epochs, callbacks):

    model.compile(

    optimizer='adam',

    loss='binary\_crossentropy',

      metrics=[

          tf.keras.metrics.BinaryAccuracy(name='accuracy'),

          tf.keras.metrics.Precision(name='precision'),

          tf.keras.metrics.Recall(name='recall')

      ]

)

    hist = model.fit(train,

                     epochs=initial\_epochs,

                     validation\_data=val,

                     callbacks=[tensorboard\_callback])

    # График функции потерь

    fig = plt.figure()

    plt.plot(hist.history['loss'], color=doom\_color, label='loss')

    plt.plot(hist.history['val\_loss'], color=ac\_color, label='val\_loss')

    fig.suptitle('Loss', fontsize=20)

    plt.legend(loc="upper left")

    plt.show()

    # График точности

    fig = plt.figure()

    plt.plot(hist.history['accuracy'], color=doom\_color, label='accuracy')

    plt.plot(hist.history['val\_accuracy'], color=ac\_color, label='val\_accuracy')

    fig.suptitle('Accuracy', fontsize=20)

    plt.legend(loc="upper left")

    plt.show()

def test\_model(model, test):

    # Расчет точности

    precision = Precision()

    recall = Recall()

    binary\_accuracy = BinaryAccuracy()

    for batch in test.as\_numpy\_iterator():

        X, y = batch

        yhat = model.predict(X)

        # Приведение вероятностей к меткам классов

        yhat\_labels = np.argmax(yhat, axis=1)

        # Обновляем метрики

        precision.update\_state(y, yhat\_labels)

        recall.update\_state(y, yhat\_labels)

        binary\_accuracy.update\_state(y, yhat\_labels)

    print(f'Precision: {precision.result().numpy():.2f}, Recall: {recall.result().numpy():.2f}, Binary Accuracy: {binary\_accuracy.result().numpy():.2f}')

Построение архитектуры более продвинутой модели в которой больше слоев, с такими же параметрачи обучения

better\_model = Sequential([

    Rescaling(1./255, input\_shape=IMG\_SHAPE),

    # Разворачиваем изображение в вектор

    Flatten(),

    Dense(512, activation='relu'),

    Dense(256, activation='relu'),

    Dense(1, activation=' sigmoid ')

])

Построение ещё более продвинутой модели с применением дропаутов, для избежания переобучения.

cool\_model = Sequential([

    Rescaling(1./255, input\_shape=IMG\_SHAPE),

    Flatten(),

    Dense(1024, activation='relu'),

    Dropout(0.4),

    Dense(512, activation='relu'),

    Dropout(0.3),

    Dense(256, activation='relu'),

    Dropout(0.2),

    Dense(1, activation=’sigmoid')

])

**7. Тестовые конфигурации сетей.**

**Первая модель.**

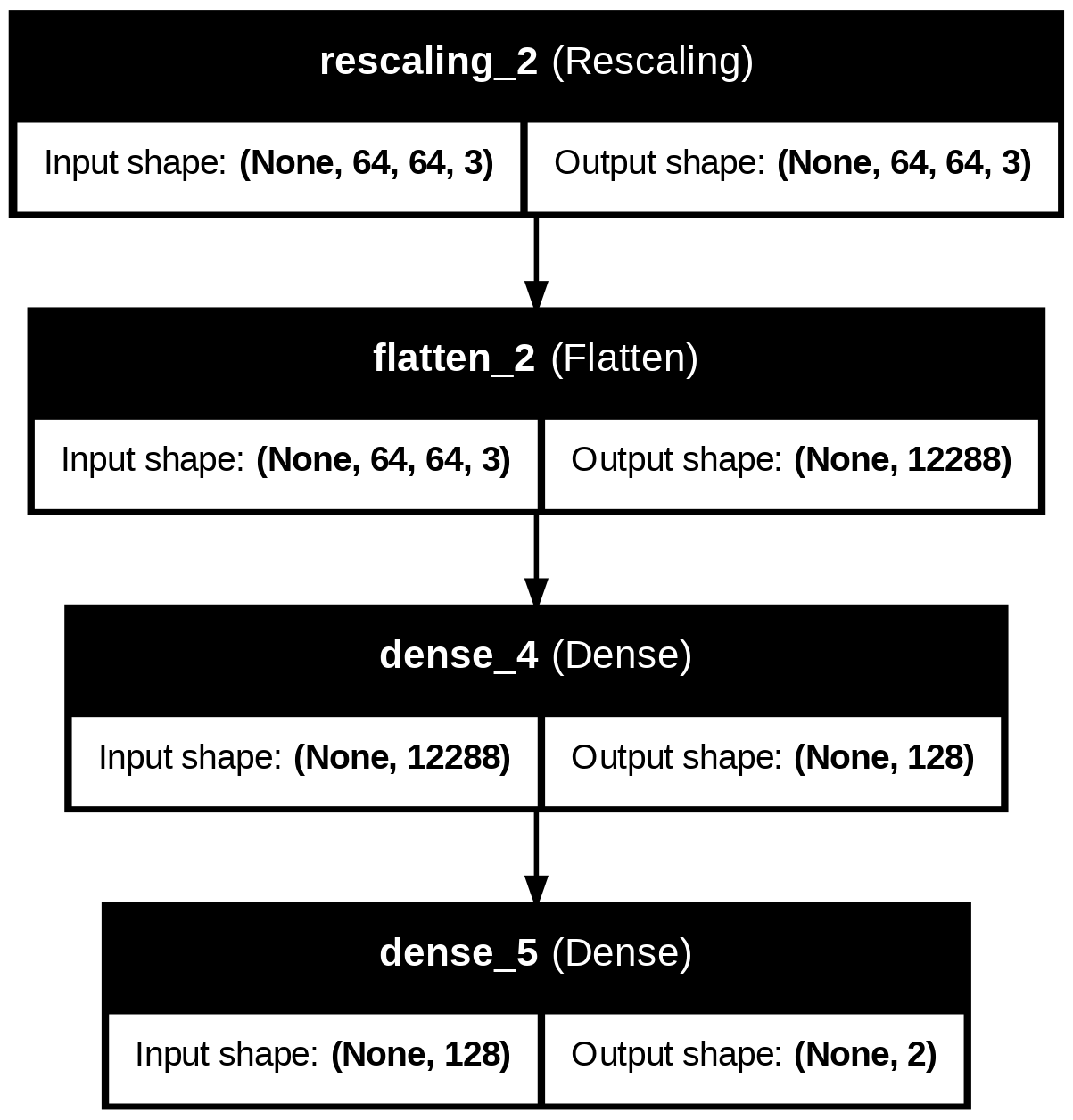


Рисунок 1 – Архитектура простой полносвязной сети

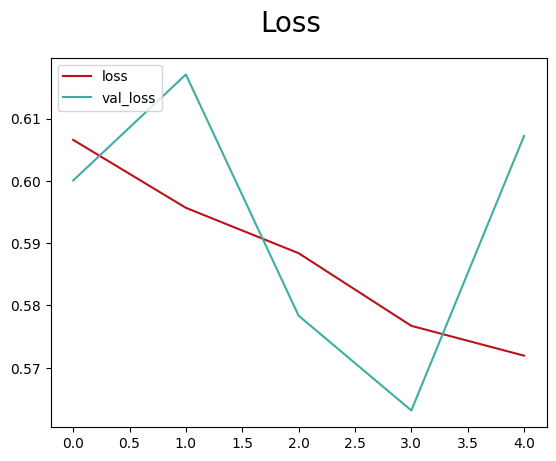


Рисунок – График функции потерь по эпохам первой модели

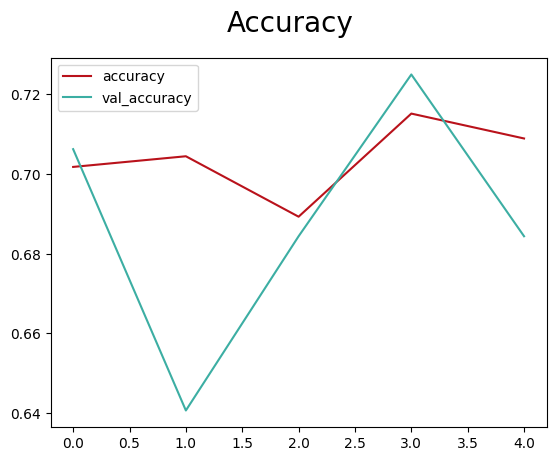


Рисунок – График расчета точности по эпохам первой модели

**Вторая модель.**

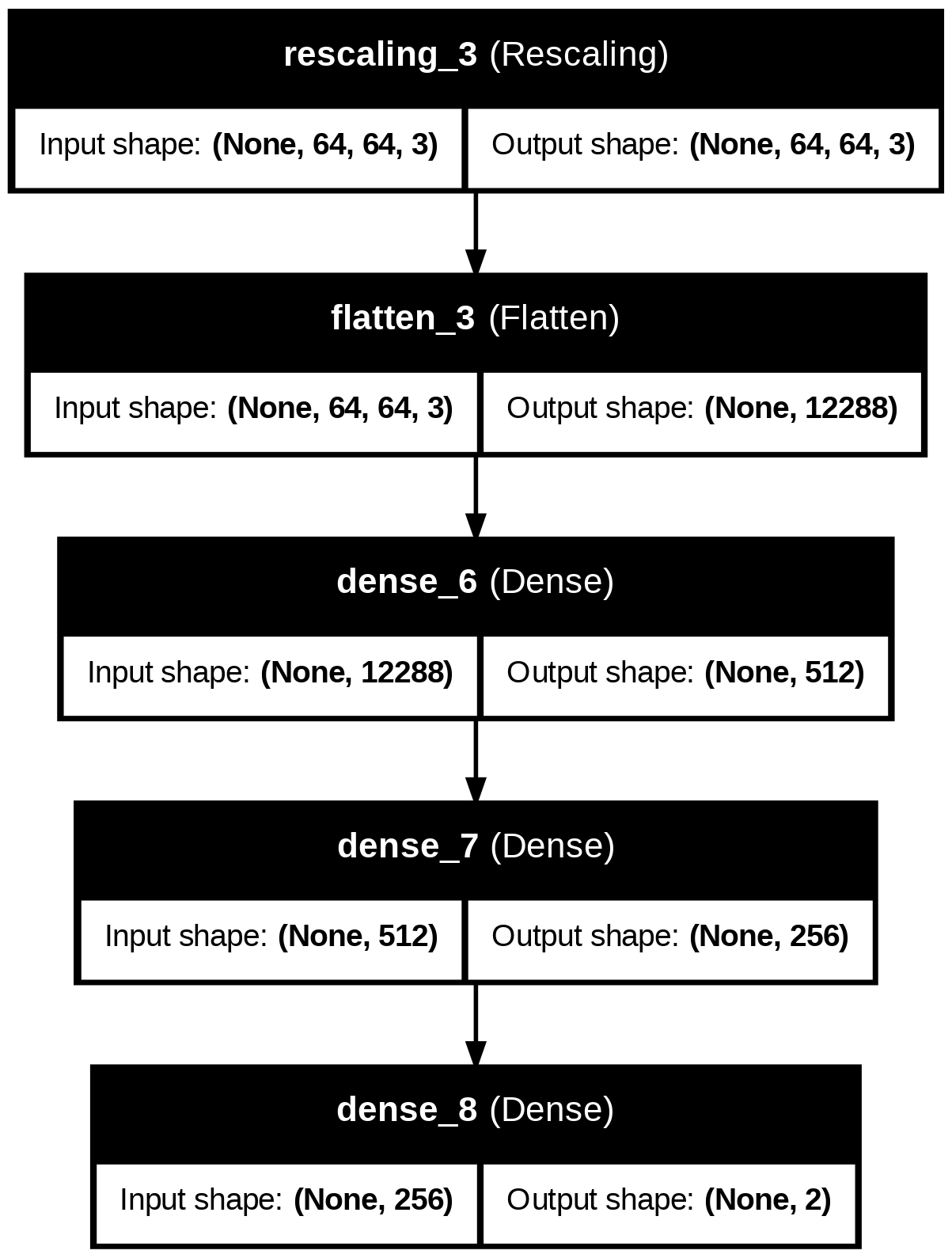


Рисунок 4 – Архитектура полносвязной сети, второй модели

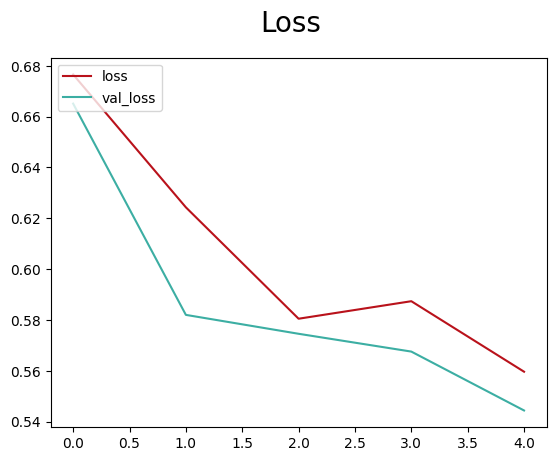


Рисунок – График функции потерь по эпохам второй модели

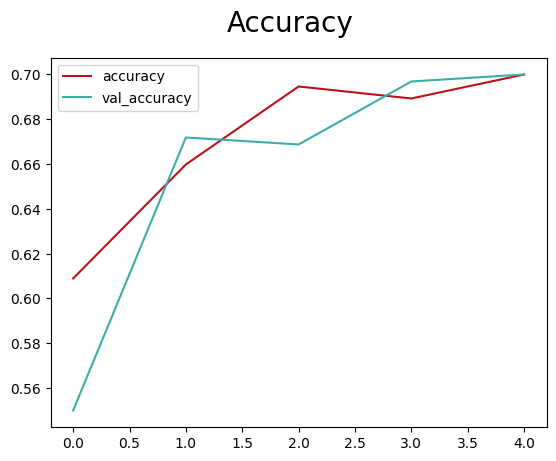


Рисунок – График расчета точности по эпохам второй модели

**Третья модель.**

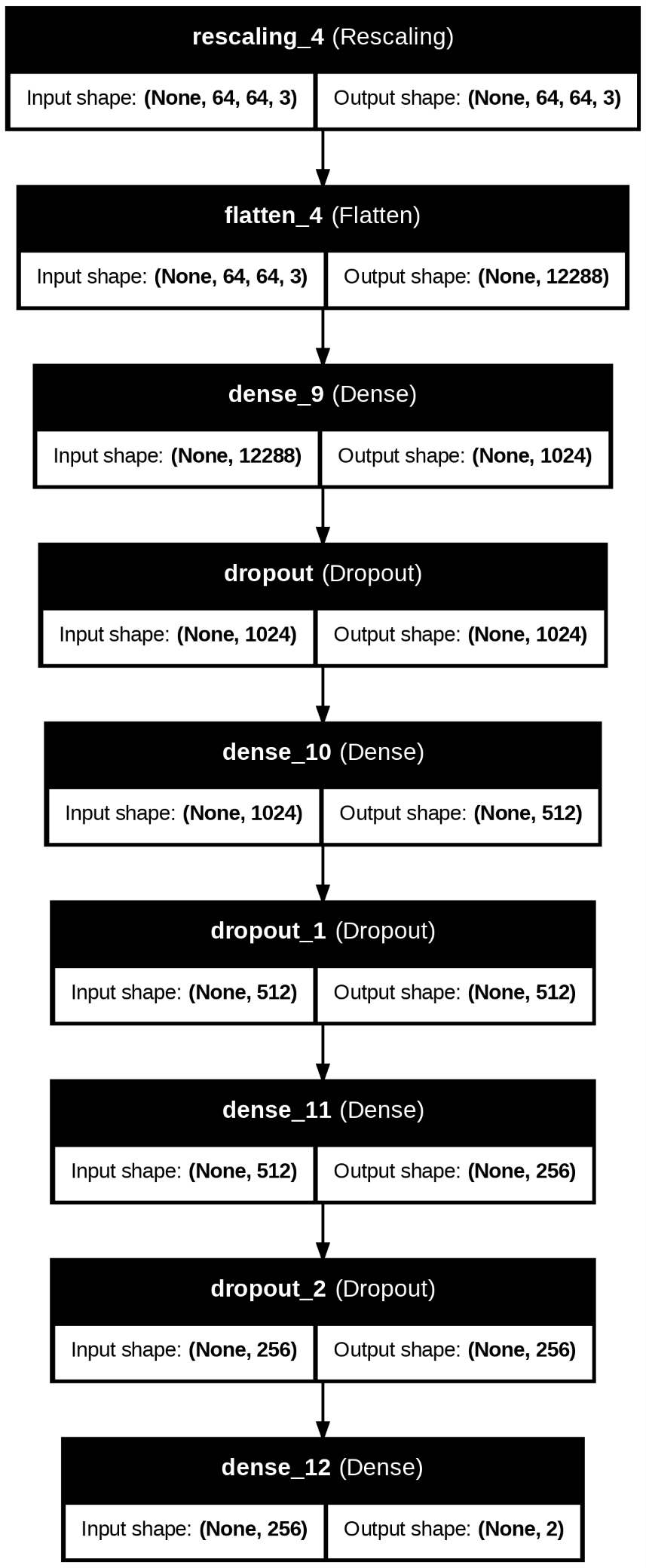


Рисунок 7 Архитектура полносвязной сети c дропаут

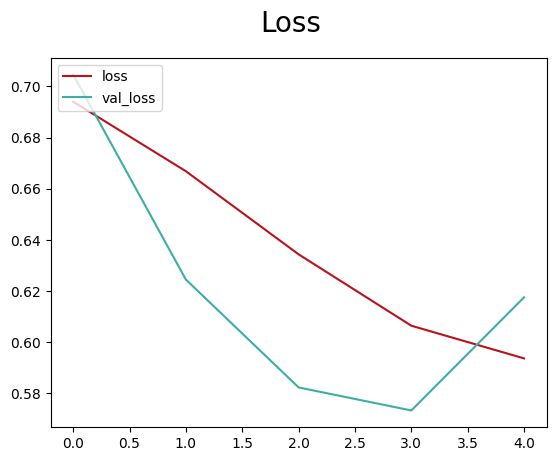


Рисунок – График функции потерь терьтей модели

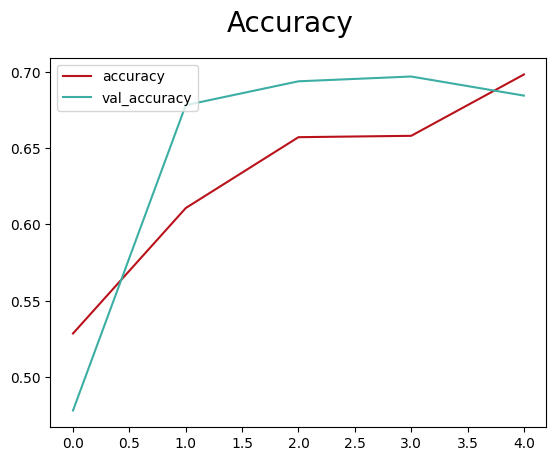


Рисунок – График точности по эпохам для третьей модели

**8. Результаты экспериментов**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Модель | Эпох | Точность (accuracy, %) | Precision (%) | Recall (%) | Время (с) |
| Простая полносвязная сеть | 5 | 77 | 75 | 75 | 308 |
| Полносвязная сеть с большим количеством слоев | 5 | 72 | 73 | 70 | 290 |
| Полносвязная сеть с большим количеством слоев и использованием дропаутов | 5 | 84 | 41 | 68 | 316 |

Выводы:

**Модель с дропаутами показывает наивысшую точность (84%), но при этом очень низкий precision (41%), что говорит о большом количестве ложных срабатываний. Такой дисбаланс может указывать на переобучение или неправильную настройку dropout-слоев.**

**Простая полносвязная сеть демонстрирует хороший баланс между precision, recall и точностью (все около 75%), что делает её наиболее стабильной из всех моделей при умеренном времени обучения.**

**Глубокая полносвязная сеть без dropout работает чуть хуже по всем метрикам и при этом не выигрывает существенно во времени, что странно, возможно произошло переобучение модели.**

Вывод: в целом увеличение количества слоёв улучшает качество модели, но может привести к переобучении. Dropout помогает предотвратить переобучение, но требует дополнительной настройки для оптимального баланса.