[Deep Learning With Keras] (CheatSheet)

Basic Operations

- Import Keras: `from tensorflow import keras`
- Create a sequential model: `model = keras.Sequential()`
- Add a dense layer: `model.add(keras.layers.Dense(units=64, activation='relu'))`
- Add an input layer: `model.add(keras.layers.Input(shape=(input_dim,)))`
- Compile a model: `model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])`
- Train a model: `model.fit(x_train, y_train, epochs=10, batch_size=32)`
- Evaluate a model: `loss, accuracy = model.evaluate(x_test, y_test)`
- Make predictions: `predictions = model.predict(x_test)`
- Save a model: `model.save('model.h5')`
- Load a model: `model = keras.models.load_model('model.h5')`

Layers

- Add a convolutional layer: `model.add(keras.layers.Conv2D(filters=32, kernel_size=(3, 3), activation='relu'))`
- Add a max pooling layer: `model.add(keras.layers.MaxPooling2D(pool_size=(2, 2)))`
- Add an average pooling layer: `model.add(keras.layers.AveragePooling2D(pool_size=(2, 2)))`
- Add a flatten layer: `model.add(keras.layers.Flatten())`
- Add a dropout layer: `model.add(keras.layers.Dropout(rate=0.5))`
- Add a batch normalization layer: `model.add(keras.layers.BatchNormalization())`
- Add a recurrent layer (SimpleRNN): `model.add(keras.layers.SimpleRNN(units=64))`
- Add a recurrent layer (LSTM): `model.add(keras.layers.LSTM(units=64))`
- Add a recurrent layer (GRU): `model.add(keras.layers.GRU(units=64))`
- Add an embedding layer: `model.add(keras.layers.Embedding(input_dim=vocab_size, output_dim=embedding_dim))`

Activation Functions

- ReLU activation: `model.add(keras.layers.Dense(units=64, activation='relu'))`
- Sigmoid activation: `model.add(keras.layers.Dense(units=1, activation='sigmoid'))`
- Tanh activation: `model.add(keras.layers.Dense(units=64, activation='tanh'))`
- Softmax activation: `model.add(keras.layers.Dense(units=num_classes, activation='softmax'))`
- LeakyReLU activation: `model.add(keras.layers.Dense(units=64, activation=keras.layers.LeakyReLU(alpha=0.1)))`
- ELU activation: `model.add(keras.layers.Dense(units=64, activation='elu'))`
- PReLU activation: `model.add(keras.layers.Dense(units=64, activation=keras.layers.PReLU()))`
- Swish activation: `model.add(keras.layers.Dense(units=64, activation=keras.activations.swish))`
- Custom activation: `model.add(keras.layers.Dense(units=64, activation=lambda x: tf.nn.relu(x) - 0.1))`

Optimizers

- SGD optimizer: `optimizer = keras.optimizers.SGD(lr=0.01, momentum=0.9)`
- Adam optimizer: `optimizer = keras.optimizers.Adam(lr=0.001)`
- RMSprop optimizer: `optimizer = keras.optimizers.RMSprop(lr=0.001)`
- Adagrad optimizer: `optimizer = keras.optimizers.Adagrad(lr=0.01)`
- Adadelta optimizer: `optimizer = keras.optimizers.Adadelta(lr=1.0)`
- Adamax optimizer: `optimizer = keras.optimizers.Adamax(lr=0.002)`
- Nadam optimizer: `optimizer = keras.optimizers.Nadam(lr=0.002)`
- Custom optimizer: `optimizer = keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999)`

Loss Functions

- Mean squared error loss: `model.compile(optimizer='adam', loss='mse')`
- Mean absolute error loss: `model.compile(optimizer='adam', loss='mae')`
- Binary crossentropy loss: `model.compile(optimizer='adam', loss='binary_crossentropy')`
- Categorical crossentropy loss: `model.compile(optimizer='adam', loss='categorical_crossentropy')`
- Sparse categorical crossentropy loss: `model.compile(optimizer='adam', loss='sparse_categorical_crossentropy')`

- KL divergence loss: model.compile(optimizer='adam', loss=keras.losses.KLDivergence())
- Huber loss: model.compile(optimizer='adam', loss=keras.losses.Huber(delta=1.0))
- Custom loss function: model.compile(optimizer='adam', loss=lambda y_true, y_pred: keras.losses.mse(y_true, y_pred) + 0.1 * keras.losses.mae(y_true, y_pred))

Metrics

- Accuracy metric: model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
- Precision metric: model.compile(optimizer='adam', loss='binary_crossentropy', metrics=[keras.metrics.Precision()])
- Recall metric: model.compile(optimizer='adam', loss='binary_crossentropy', metrics=[keras.metrics.Recall()])
- F1 score metric: model.compile(optimizer='adam', loss='binary_crossentropy', metrics=[keras.metrics.F1Score()])
- AUC metric: model.compile(optimizer='adam', loss='binary_crossentropy', metrics=[keras.metrics.AUC()])
- Custom metric: model.compile(optimizer='adam', loss='mse', metrics=[lambda y_true, y_pred: keras.backend.mean(keras.backend.abs(y_true - y_pred))])

Callbacks

- Early stopping: early_stopping = keras.callbacks.EarlyStopping(monitor='val_loss', patience=5)
- Model checkpoint: model_checkpoint = keras.callbacks.ModelCheckpoint('best_model.h5', save_best_only=True)
- Learning rate scheduler: lr_scheduler = keras.callbacks.LearningRateScheduler(lambda epoch: 0.001 * 0.1 ** (epoch // 10))
- Tensorboard: tensorboard = keras.callbacks.TensorBoard(log_dir='logs')
- CSV logger: csv_logger = keras.callbacks.CSVLogger('training.log')
- Custom callback: class CustomCallback(keras.callbacks.Callback): def on_epoch_end(self, epoch, logs=None): print(f'Epoch {epoch}: Loss={logs["loss"]}, Accuracy={logs["accuracy"]}')

Regularization

- L1 regularization: `model.add(keras.layers.Dense(units=64, activation='relu', kernel_regularizer=keras.regularizers.l1(0.01)))`
- L2 regularization: `model.add(keras.layers.Dense(units=64, activation='relu', kernel_regularizer=keras.regularizers.l2(0.01)))`
- L1 and L2 regularization: `model.add(keras.layers.Dense(units=64, activation='relu', kernel_regularizer=keras.regularizers.l1_l2(l1=0.01, l2=0.01)))`
- Dropout regularization: `model.add(keras.layers.Dropout(rate=0.5))`
- Gaussian noise regularization: `model.add(keras.layers.GaussianNoise(stddev=0.1))`
- Activity regularization: `model.add(keras.layers.Dense(units=64, activation='relu', activity_regularizer=keras.regularizers.l2(0.01)))`
- Custom regularizer: `model.add(keras.layers.Dense(units=64, activation='relu', kernel_regularizer=keras.regularizers.l1(0.01) + keras.regularizers.l2(0.01)))`

Initializers

- Glorot uniform initializer: `model.add(keras.layers.Dense(units=64, kernel_initializer='glorot_uniform'))`
- Glorot normal initializer: `model.add(keras.layers.Dense(units=64, kernel_initializer='glorot_normal'))`
- He uniform initializer: `model.add(keras.layers.Dense(units=64, kernel_initializer='he_uniform'))`
- He normal initializer: `model.add(keras.layers.Dense(units=64, kernel_initializer='he_normal'))`
- Orthogonal initializer: `model.add(keras.layers.Dense(units=64, kernel_initializer='orthogonal'))`
- Identity initializer: `model.add(keras.layers.Dense(units=64, kernel_initializer='identity'))`
- Constant initializer: `model.add(keras.layers.Dense(units=64, kernel_initializer=keras.initializers.Constant(value=0.1)))`
- Truncated normal initializer: `model.add(keras.layers.Dense(units=64, kernel_initializer=keras.initializers.TruncatedNormal(mean=0.0, stddev=0.1)))`

Preprocessing

- Image data generator: `datagen = keras.preprocessing.image.ImageDataGenerator(rotation_range=20, width_shift_range=0.1, height_shift_range=0.1, zoom_range=0.1)`

- Text tokenizer: tokenizer = keras.preprocessing.text.Tokenizer(num_words=10000)
- Sequence padding: padded_sequences = keras.preprocessing.sequence.pad_sequences(sequences, maxlen=100)
- One-hot encoding: one_hot = keras.utils.to_categorical(labels)
- Train-test split: x_train, x_test, y_train, y_test = keras.preprocessing.text.train_test_split(x, y, test_size=0.2)
- Normalization: normalized_data = keras.utils.normalize(data)
- Standardization: standardized_data = (data - np.mean(data)) / np.std(data)

Transfer Learning

- VGG16 model: base_model = keras.applications.VGG16(weights='imagenet', include_top=False)
- ResNet50 model: base_model = keras.applications.ResNet50(weights='imagenet', include_top=False)
- InceptionV3 model: base_model = keras.applications.InceptionV3(weights='imagenet', include_top=False)
- Xception model: base_model = keras.applications.Xception(weights='imagenet', include_top=False)
- MobileNet model: base_model = keras.applications.MobileNet(weights='imagenet', include_top=False)
- DenseNet model: base_model = keras.applications.DenseNet121(weights='imagenet', include_top=False)
- NASNet model: base_model = keras.applications.NASNetLarge(weights='imagenet', include_top=False)
- EfficientNet model: base_model = keras.applications.EfficientNetB0(weights='imagenet', include_top=False)

Model Evaluation

- Confusion matrix: confusion_matrix = keras.metrics.confusion_matrix(y_true, y_pred)
- Classification report: classification_report = keras.metrics.classification_report(y_true, y_pred)
- Precision-recall curve: precision, recall, thresholds = keras.metrics.precision_recall_curve(y_true, y_pred)
- ROC curve: fpr, tpr, thresholds = keras.metrics.roc_curve(y_true, y_pred)

- Learning curves: `history = model.fit(x_train, y_train, validation_data=(x_val, y_val), epochs=20); plt.plot(history.history['loss']); plt.plot(history.history['val_loss'])`
- Visualization of intermediate activations: `activations = keras.backend.function([model.layers[0].input], [model.layers[i].output])`
- Visualization of convolutional filters: `filters, biases = model.layers[i].get_weights()`
- Visualization of saliency maps: `saliency_map = keras.backend.gradients(model.output, model.input)[0]`

Model Saving and Loading

- Save model architecture: `model_json = model.to_json()`
- Load model architecture: `model = keras.models.model_from_json(model_json)`
- Save model weights: `model.save_weights('model_weights.h5')`
- Load model weights: `model.load_weights('model_weights.h5')`
- Save entire model: `model.save('model.h5')`
- Load entire model: `model = keras.models.load_model('model.h5')`

Advanced Techniques

- Custom layer: `class CustomLayer(keras.layers.Layer): def __init__(self, units=32): super(CustomLayer, self).__init__(); self.units = units; def build(self, input_shape): self.w = self.add_weight(shape=(input_shape[-1], self.units), initializer='random_normal', trainable=True); def call(self, inputs): return keras.backend.dot(inputs, self.w)`
- Custom model: `class CustomModel(keras.Model): def __init__(self): super(CustomModel, self).__init__(); self.dense1 = keras.layers.Dense(64, activation='relu'); self.dense2 = keras.layers.Dense(10, activation='softmax'); def call(self, inputs): x = self.dense1(inputs); return self.dense2(x)`
- Gradient clipping: `optimizer = keras.optimizers.Adam(clipvalue=0.5)`
- Learning rate scheduling: `lr_schedule = keras.optimizers.schedules.ExponentialDecay(initial_learning_rate=0.01, decay_steps=10000, decay_rate=0.9)`
- Mixed precision training: `keras.mixed_precision.set_global_policy('mixed_float16')`
- Distributed training with TensorFlow: `strategy = tf.distribute.MirroredStrategy(); with strategy.scope(): model = create_model()`

- Model subclassing: `class SubclassModel(keras.Model): def __init__(self): super(SubclassModel, self).__init__(); self.conv1 = keras.layers.Conv2D(32, 3, activation='relu'); self.flatten = keras.layers.Flatten(); self.dense1 = keras.layers.Dense(128, activation='relu'); self.dense2 = keras.layers.Dense(10, activation='softmax'); def call(self, inputs): x = self.conv1(inputs); x = self.flatten(x); x = self.dense1(x); return self.dense2(x)`
- Hyperparameter tuning with Keras Tuner: `tuner = keras_tuner.RandomSearch(build_model, objective='val_accuracy', max_trials=10); tuner.search(x_train, y_train, epochs=10, validation_data=(x_val, y_val))`
- Tensorboard visualization: `tensorboard_callback = keras.callbacks.TensorBoard(log_dir='./logs', histogram_freq=1)`

Generative Models

- Variational Autoencoder (VAE): `encoder = keras.Sequential([keras.layers.Dense(64, activation='relu'), keras.layers.Dense(32, activation='relu')]); decoder = keras.Sequential([keras.layers.Dense(64, activation='relu'), keras.layers.Dense(original_dim, activation='sigmoid')]); vae = keras.Model(inputs=encoder.input, outputs=decoder(encoder.output))`
- Generative Adversarial Network (GAN): `generator = keras.Sequential([keras.layers.Dense(128, activation='relu'), keras.layers.Dense(original_dim, activation='tanh')]); discriminator = keras.Sequential([keras.layers.Dense(128, activation='relu'), keras.layers.Dense(1, activation='sigmoid')]); gan = keras.Sequential([generator, discriminator])`
- Conditional GAN (cGAN): `cgan = keras.Sequential([generator, discriminator]); cgan.compile(loss=['binary_crossentropy', 'sparse_categorical_crossentropy'], optimizer=keras.optimizers.Adam(0.0002, 0.5), metrics=['accuracy'])`
- Cycle GAN: `cyclegan = keras.Model(inputs=[real_image, target_image], outputs=[fake_output, rec_input, identity_output])`
- Pix2Pix: `pix2pix = keras.Model(inputs=input_image, outputs=generator(input_image))`

Model Interpretability

- Sensitivity analysis: `sensitivity_map = keras.backend.gradients(model.output, model.input)[0]`

- Layer activation visualization: `intermediate_output = keras.backend.function([model.input], [model.layers[i].output])`
- Gradient-weighted Class Activation Mapping (Grad-CAM): `grad_cam = keras.backend.gradients(model.output, model.layers[i].output)[0]`
- SHAP values: `explainer = shap.DeepExplainer(model, x_train[:100]); shap_values = explainer.shap_values(x_test[:10])`
- LIME: `explainer = lime_image.LimeImageExplainer(); explanation = explainer.explain_instance(x_test[0], model.predict, top_labels=5, hide_color=0, num_samples=1000)`

Model Optimization

- Model pruning: `pruning_params = {'pruning_schedule': keras.optimizers.schedules.PolynomialDecay(initial_sparsity=0.5, final_sparsity=0.9, power=3, begin_step=0, end_step=10000)}; model = keras.models.Sequential([keras.layers.Dense(64, activation='relu'), keras.layers.Dense(10, activation='softmax')]); model = keras.models.Sequential([keras.layers.Prune(model.layers[0], **pruning_params), model.layers[1]])`
- Knowledge distillation: `teacher_model = create_teacher_model(); student_model = create_student_model(); distilled_model = keras.Model(inputs=student_model.input, outputs=student_model.output); distilled_model.compile(optimizer=keras.optimizers.Adam(), loss=keras.losses.KLDivergence())`
- Quantization: `quantize_model = tfmot.quantization.keras.quantize_model(model)`
- Clustering: `clustered_model = tfmot.clustering.keras.cluster_model(model, number_of_clusters=16, cluster_centroids_init=tfmot.clustering.keras.CentroidInitialization.LINEAR)`
- Weight sharing: `shared_weights_model = tfmot.clustering.keras.share_weights(model, tfmot.clustering.keras.SharedWeights.CENTROID_INITIALIZATION)`

Model Deployment

- Convert to TensorFlow Lite: `converter = tf.lite.TFLiteConverter.from_keras_model(model); tflite_model = converter.convert()`
- Convert to TensorFlow.js: `tfjs.converters.save_keras_model(model, 'path/to/save')`

- Convert to ONNX: `onnx_model = onnx_tf.keras2onnx.convert_keras(model, model.name)`
- Serve model with TensorFlow Serving: `model.save('model_dir'); os.system('tensorflow_model_server --rest_api_port=8501 --model_name=model --model_base_path=model_dir')`
- Deploy model on mobile devices: `converter = tf.lite.TFLiteConverter.from_keras_model(model); tflite_model = converter.convert(); open('model.tflite', 'wb').write(tflite_model)`