Keras

AI+X
INSTITUTE FOR ARTIFICIAL INTELLIGENCE

Deep Learning APIs

Keras

TensorFlow

PyTorch

Figure 1. Deep learning APIs.

Keras

Keras is an API designed for human beings, not machines. Keras follows best practices for reducing cognitive load: it offers consistent & simple APIs, it minimizes the number of user actions required for common use cases, and it provides clear & actionable error messages. It also has extensive documentation and developer guides.

- Please provide source

- **Documentation:** https://www.tensorflow.org/api_docs/python/tf/keras/
- **Installation:** pip install tensorflow

Figure 2. Logo for Keras deep learning API.

Data Manipulation

We can use NumPy!

Network Implementation - Input

https://www.tensorflow.org/api_docs/python/tf/keras/Input

```
tf.keras.Input(
        shape=None,
        batch_size=None,
        name=None,
        dtype=None,
        sparse=None,
        tensor=None,
        ragged=None,
        type_spec=None,
        **kwargs
)
```

Args	
shape	A shape tuple (integers), not including the batch size. For instance, shape=(32,) indicates that the expected input will be batches of 32-dimensional vectors. Elements of this tuple can be None; 'None' elements represent dimensions where the shape is not known.
batch_size	optional static batch size (integer).
name	An optional name string for the layer. Should be unique in a model (do not reuse the same name twice). It will be autogenerated if it isn't provided.
dtype	The data type expected by the input, as a string (float32, float64, int32...)
sparse	A boolean specifying whether the placeholder to be created is sparse. Only one of 'ragged' and 'sparse' can be True. Note that, if sparse is False, sparse tensors can still be passed into the input - they will be densified with a default value of 0.
tensor	Optional existing tensor to wrap into the Input layer. If set, the layer will use the tf.TypeSpec of this tensor rather than creating a new placeholder tensor.
ragged	A boolean specifying whether the placeholder to be created is ragged. Only one of 'ragged' and 'sparse' can be True. In this case, values of 'None' in the 'shape' argument represent ragged dimensions. For more information about RaggedTensors, see this guide.
type_spec	A tf.TypeSpec object to create the input placeholder from. When provided, all other args except name must be None.
**kwargs	deprecated arguments support. Supports batch_shape and batch_input_shape.

Returns
A tensor.

Figure 3. Network implementation - Input.

Network Implementation – Dense Layer

https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dense

```
tf.keras.layers.Dense(
        units,
        activation=None,
        use_bias=True,
        kernel_initializer='glorot_uniform',
        bias_initializer='zeros',
        kernel_regularizer=None,
        bias_regularizer=None,
        activity_regularizer=None,
        kernel_constraint=None,
        bias_constraint=None,
        **kwargs
)
```

Args	
`units`	Positive integer, dimensionality of the output space.
`activation`	Activation function to use. If you don't specify anything, no activation is applied (ie. "linear" activation: `a(x) = x`).
`use_bias`	Boolean, whether the layer uses a bias vector.
`kernel_initializer`	Initializer for the `kernel` weights matrix.
`bias_initializer`	Initializer for the bias vector.
`kernel_regularizer`	Regularizer function applied to the `kernel` weights matrix.
`bias_regularizer`	Regularizer function applied to the bias vector.
`activity_regularizer`	Regularizer function applied to the output of the layer (its "activation").
`kernel_constraint`	Constraint function applied to the `kernel` weights matrix.
`bias_constraint`	Constraint function applied to the bias vector.

Input shape

N-D tensor with shape: `(batch_size, ..., input_dim)`. The most common situation would be a 2D input with shape `(batch_size, input_dim)`.

Output shape

N-D tensor with shape: `(batch_size, ..., units)`. For instance, for a 2D input with shape `(batch_size, input_dim)`, the output would have shape `(batch_size, units)`.

Figure 4. Network implementation – Dense layer.

Network Implementation - ReLU

https://www.tensorflow.org/api_docs/python/tf/keras/activations/relu

```
tf.keras.activations.relu(
        x,
        alpha=0.0,
        max_value=None,
        threshold=0.0
)
```

Args	
x	Input tensor or variable.
alpha	A float that governs the slope for values lower than the threshold.
max_value	A float that sets the saturation threshold (the largest value the function will return).
threshold	A float giving the threshold value of the activation function below which values will be damped or set to zero.

Returns
A Tensor representing the input tensor, transformed by the relu activation function. Tensor will be of the same shape and dtype of input x.

Figure 5. Network implementation – ReLU.

Network Implementation – Sequential

https://www.tensorflow.org/api_docs/python/tf/keras/Sequential

```
tf.keras.Sequential(
        layers=None,
    name=None
)
```

Methods:

```
    add(
        layer
    )
```

Args	
`layers`	Optional list of layers to add to the model.
`name`	Optional name for the model.

Args	
`layer`	layer instance.

Raises	
`TypeError`	If `layer` is not a layer instance.
`ValueError`	In case the `layer` argument does not know its input shape.
`ValueError`	In case the `layer` argument has multiple output tensors, or is already connected somewhere else (forbidden in `Sequential` models).

Figure 6. Network implementation – Sequential.

Network Implementation - MLP

```
# create new sequential model
model = tf.keras.models.Sequential()

# specify input size
model.add(tf.keras.Input(shape=(2,)))

# create a dense layer with 8 output values and ReLU activation
model.add(tf.keras.layers.Dense(8, activation=tf.keras.activations.relu))

# create a dense layer with 1 output value and no activation
model.add(tf.keras.layers.Dense(1))
```

Network Implementation - Model

https://www.tensorflow.org/api_docs/python/tf/keras/Model

```
tf.keras.Model(
        inputs,
        outputs,
        name,
        **kwargs
)
```

Args	
inputs	The input(s) of the model: a keras.Input object or list of keras.Input objects.
outputs	The output(s) of the model. See Functional API example below.
name	String, the name of the model.

Figure 7. Network implementation – Model.

Network Implementation - MLP

```
# specify the input size
input = tf.keras.Input(shape=(2,))

# create a dense layer with 8 output values and ReLU activation
hidden = tf.keras.layers.Dense(8, activation=tf.keras.activations.relu)(input)

# create a dense layer with 1 output value and no activation
output = tf.keras.layers.Dense(1)(hidden)

# create new model
model = tf.keras.models.Model(inputs=input, outputs=output)
```

Knowledge Check 1

In the network below, why do we use a ReLU activation in the first layer, but not in the second one?

```
input = tf.keras.Input(shape=(2,))
hidden = tf.keras.layers.Dense(8,
activation=tf.keras.activations.relu)(input)
output = tf.keras.layers.Dense(1)(hidden)
model = tf.keras.models.Model(
  inputs=input,
  outputs=output
)
```

- **A** Because a nonlinear activation function in between linear (Dense) layers gives the network the ability to solve nonlinear problems
- **B** Because a ReLU activation in the second layer would limit the network output to non-negative values
- **C** Because it is not common to use ReLU as the final activation function of a network
- **D** All the above

Training – MSE Loss

https://www.tensorflow.org/api_docs/python/tf/keras/losses/MSE

```
tf.keras.metrics.mean_squared_error(
    y_true,
    y_pred
)
```

Args	
y_true	Ground truth values. shape = [batch_size, d0, .. dN].
y_pred	The predicted values. shape = [batch_size, d0, .. dN].

Returns
Mean squared error values. shape = [batch_size, d0, .. dN-1].

Figure 8. Training – MSE loss.

Training – SGD Optimizer

https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/SGD

```
tf.keras.optimizers.SGD(
        learning_rate=0.01,
        momentum=0.0,
        nesterov=False,
        name='SGD',
        **kwargs
)
```

Args	
learning_rate	A Tensor, floating point value, or a schedule that is a tf.keras.optimizers.schedules.LearningRateSchedule, or a callable that takes no arguments and returns the actual value to use. The learning rate. Defaults to 0.01.
momentum	float hyperparameter >= 0 that accelerates gradient descent in the relevant direction and dampens oscillations. Defaults to 0, i.e., vanilla gradient descent.
nesterov	boolean. Whether to apply Nesterov momentum. Defaults to False.
name	Optional name prefix for the operations created when applying gradients. Defaults to "SGD".
**kwargs	keyword arguments. Allowed arguments are clipvalue, clipnorm, global_clipnorm. If clipvalue (float) is set, the gradient of each weight is clipped to be no higher than this value. If clipnorm (float) is set, the gradient of each weight is individually clipped so that its norm is no higher than this value. If global_clipnorm (float) is set the gradient of all weights is clipped so that their global norm is no higher than this value.

Figure 9. Training – SGD optimizer.

Training – Model Compile

https://www.tensorflow.org/api_docs/python/tf/keras/Model#compile

```
compile(
        optimizer='rmsprop',
        loss=None,
        metrics=None,
        loss_weights=None,
        weighted_metrics=None,
        run_eagerly=None,
        steps_per_execution=None,
        jit_compile=None,
        **kwargs
)
```

Args	
optimizer	String (name of optimizer) or optimizer instance. See `tf.keras.optimizers`.
loss	Loss function. May be a string (name of loss function), or a `tf.keras.losses.Loss` instance. See `tf.keras.losses`. A loss function is any callable with the signature `loss = fn(y_true, y_pred)`, where `y_true` are the ground truth values, and `y_pred` are the model's predictions. `y_true` should have shape `(batch_size, d0, .. dN)` (except in the case of sparse loss functions such as sparse categorical crossentropy which expects integer arrays of shape `(batch_size, d0, .. dN-1)`). `y_pred` should have shape `(batch_size, d0, .. dN)`. The loss function should return a float tensor. If a custom `Loss` instance is used and reduction is set to `None`, return value has shape `(batch_size, d0, .. dN-1)` i.e. per-sample or per-timestep loss values; otherwise, it is a scalar. If the model has multiple outputs, you can use a different loss on each output by passing a dictionary or a list of losses. The loss value that will be minimized by the model will then be the sum of all individual losses, unless `loss_weights` is specified.
metrics	List of metrics to be evaluated by the model during training and testing. Each of this can be a string (name of a built-in function), function or a `tf.keras.metrics.Metric` instance. See `tf.keras.metrics`. Typically you will use `metrics=['accuracy']`. A function is any callable with the signature `result = fn(y_true, y_pred)`. To specify different metrics for different outputs of a multi-output model, you could also pass a dictionary, such as `metrics={'output_a':'accuracy', 'output_b':['accuracy', 'mse']}`. You can also pass a list to specify a metric or a list of metrics for each output, such as `metrics=[['accuracy'], ['accuracy', 'mse']]` or `metrics=['accuracy', ['accuracy', 'mse']]`. When you pass the strings 'accuracy' or 'acc', we convert this to one of `tf.keras.metrics.BinaryAccuracy`, `tf.keras.metrics.CategoricalAccuracy`, `tf.keras.metrics.SparseCategoricalAccuracy` based on the loss function used and the model output shape. We do a similar conversion for the strings 'crossentropy' and 'ce' as well. The metrics passed here are evaluated without sample weighting; if you would like sample weighting to apply, you can specify your metrics via the `weighted_metrics` argument instead.

Figure 10. Training – Model compile.

Training – Model Fit

https://www.tensorflow.org/api_docs/python/tf/keras/Model#fit

```
fit(
        x=None,
        y=None,
        batch_size=None,
        epochs=1,
        verbose='auto',
        callbacks=None,
        validation_split=0.0,
        validation_data=None,
        shuffle=True,
        class_weight=None,
        sample_weight=None,
        initial_epoch=0,
        steps_per_epoch=None,
        validation_steps=None,
        validation_batch_size=None,
        validation_freq=1,
        max_queue_size=10,
        workers=1,
        use_multiprocessing=False
)
```

Args	
x	Input data. It could be: • A Numpy array (or array-like), or a list of arrays (in case the model has multiple inputs). • A TensorFlow tensor, or a list of tensors (in case the model has multiple inputs). • A dict mapping input names to the corresponding array/tensors, if the model has named inputs. • A `tf.data` dataset. Should return a tuple of either `(inputs, targets)` or `(inputs, targets, sample_weights)`. • A generator or `keras.utils.Sequence` returning `(inputs, targets)` or `(inputs, targets, sample_weights)`. • A `tf.keras.utils.experimental.DatasetCreator`, which wraps a callable that takes a single argument of type `tf.distribute.InputContext`, and returns a `tf.data.Dataset`. `DatasetCreator` should be used when users prefer to specify the per-replica batching and sharding logic for the `Dataset`. See `tf.keras.utils.experimental.DatasetCreator` doc for more information. A more detailed description of unpacking behavior for iterator types (Dataset, generator, Sequence) is given below. If these include `sample_weights` as a third component, note that sample weighting applies to the `weighted_metrics` argument but not the `metrics` argument in `compile()`. If using `tf.distribute.experimental.ParameterServerStrategy`, only `DatasetCreator` type is supported for `x`.
y	Target data. Like the input data `x`, it could be either Numpy array(s) or TensorFlow tensor(s). It should be consistent with `x` (you cannot have Numpy inputs and tensor targets, or inversely). If `x` is a dataset, generator, or `keras.utils.Sequence` instance, `y` should not be specified (since targets will be obtained from `x`).
batch_size	Integer or `None`. Number of samples per gradient update. If unspecified, `batch_size` will default to 32. Do not specify the `batch_size` if your data is in the form of datasets, generators, or `keras.utils.Sequence` instances (since they generate batches).
epochs	Integer. Number of epochs to train the model. An epoch is an iteration over the entire `x` and `y` data provided (unless the `steps_per_epoch` flag is set to something other than None). Note that in conjunction with `initial_epoch`, `epochs` is to be understood as "final epoch". The model is not trained for a number of iterations given by `epochs`, but merely until the epoch of index `epochs` is reached.

Figure 11. Training – Model fit.

Training Code

```
# define the optimizer that will be used to apply gradient-based updates
optimizer = tf.keras.optimizers.SGD(learning_rate=0.5)

# compile the model with the chosen loss function and optimizer
model.compile(loss=tf.keras.metrics.mean_squared_error, optimizer=optimizer)

# train the model with data samples 'X' and their expected output 'y'
# for gradient descent:
#   - batch size should be the number of training samples
model.fit(X, y, batch_size=len(X), epochs=2048)
```

Common Practice #4: Train Using Mini-Batches

- Gradient descent uses the entire data in every training step
- Stochastic Gradient Descent (SGD) computes gradients from a randomly chosen subset of the data in every training step
- More updates, faster convergence
- Batch size is a hyperparameter
    - Common practice #1 - Hyperparameter tuning

Training Code

```
# define the optimizer that will be used to apply gradient-based updates
optimizer = tf.keras.optimizers.SGD(learning_rate=0.5)

# compile the model with the chosen loss function and optimizer
model.compile(loss=tf.keras.metrics.mean_squared_error, optimizer=optimizer)

# train the model with data samples 'X' and their expected output 'y'
# for stochastic gradient descent:
#    - batch size should be smaller than the number of training samples
#    - batch_size <<< len(X)
model.fit(X, y, batch_size=32, epochs=2048)
```

Knowledge Check 2

Why should we use SGD instead of gradient descent?

I) Unlike gradient descent, SGD works for huge datasets, as it only processes small mini-batches at a time.

II) Weight updates for SGD are based on mini-batches, which makes them more accurate towards the global minimum than using updates based on all training samples as a batch.

III) SGD converges faster than gradient descent because it can perform multiple weight updates in a single pass over the training samples.

A. I and II

B. I and III

C. II and III

D. I, II, and III

Final Activation & Loss Functions

Problem type	Output type	Final activation function	Loss function
Regression	Numerical value	None	Mean Squared Error
Classification	Binary outcome	Sigmoid	Binary Cross-entropy
Classification	Single label, multiple classes	Softmax	Categorical Cross-entropy

Table 1. Final activation and loss functions.

Binary Classification

- Am I overweight?
  - Yes, if BMI ≥ 25
  - No, otherwise
- Sigmoid activation
  - $\sigma(z) = 1 / (1 + e^{-z})$
- Binary cross entropy loss
  - $\mathcal{L} = -[p \log q + (1 - p) \log(1 - q)]$
  - p →label / q → output

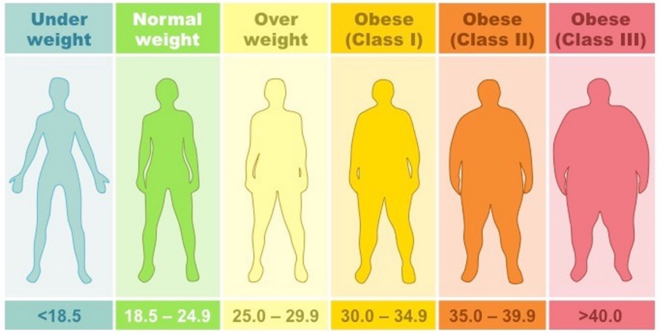

Figure 12. Binary classification.

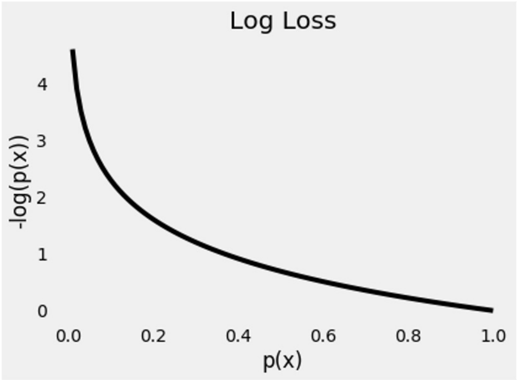

Figure 13. Binary classification.

Network Implementation - Sigmoid

https://www.tensorflow.org/api_docs/python/tf/keras/activations/sigmoid

```
tf.keras.activations.sigmoid(
    x
)
```

Args	
x	Input tensor.

Returns
Tensor with the sigmoid activation: `1 / (1 + exp(-x))`.

Figure 14. Network implementation – Sigmoid.

Network Implementation – Binary Cross-entropy

https://www.tensorflow.org/api_docs/python/tf/keras/metrics/binary_crossentropy

```
tf.keras.metrics.binary_crossentropy(
    y_true,
    y_pred,
    from_logits=False,
    label_smoothing=0.0,
    axis=-1
)
```

Args	
y_true	Ground truth values. shape = [batch_size, d0, .. dN].
y_pred	The predicted values. shape = [batch_size, d0, .. dN].
from_logits	Whether y_pred is expected to be a logits tensor. By default, we assume that y_pred encodes a probability distribution.
label_smoothing	Float in [0, 1]. If > 0 then smooth the labels by squeezing them towards 0.5 That is, using 1. - 0.5 * label_smoothing for the target class and 0.5 * label_smoothing for the non-target class.
axis	The axis along which the mean is computed. Defaults to -1.

Returns
Binary crossentropy loss value. shape = [batch_size, d0, .. dN-1].

Figure 15. Network implementation – Binary cross-entropy.

Binary Classification

```
# create model
model = tf.keras.models.Sequential()
model.add(tf.keras.Input(shape=(2,)))
model.add(tf.keras.layers.Dense(8, activation=tf.keras.activations.relu))
model.add(tf.keras.layers.Dense(1, activation=tf.keras.activations.sigmoid))

# training
optimizer = tf.keras.optimizers.SGD(lr=0.1)
model.compile(loss=tf.keras.metrics.binary_crossentropy, optimizer=optimizer)
model.fit(X, y, batch_size=32, epochs=512)
```

Multiclass Classification

- What is my weight category?
  - Underweight: BMI < 18.5
  - Normal: 18.5 ≤ BMI < 25
  - Overweight: 25 ≤ BMI < 30
  - Obese: 30 ≤ BMI
- Softmax activation
  - $\varphi(z_j) = e^{z[j]} / \sum_i e^{z[i]}$
- Categorical cross entropy loss
  - $\mathcal{L} = -\Sigma_i p_i \log q_i$
  - p →label / q → output

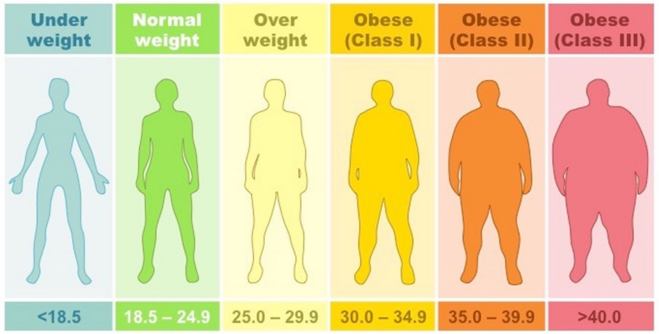

Figure 16. Multiclass classification.

Network Implementation - Softmax

https://www.tensorflow.org/api_docs/python/tf/keras/activations/softmax

```
tf.keras.activations.softmax(
    x,
    axis=-1
)
```

Args	
x	Input tensor.
axis	Integer, axis along which the softmax normalization is applied.

Returns
Tensor, output of softmax transformation (all values are non-negative and sum to 1).

Figure 17. Network implementation – Softmax.

Network Implementation - Cross-entropy

https://www.tensorflow.org/api_docs/python/tf/keras/metrics/sparse_categorical_crossentropy

```
tf.keras.metrics.sparse_categorical_crossentropy(
    y_true,
    y_pred,
    from_logits=False,
    axis=-1,
    ignore_class=None
)
```

Args	
y_true	Ground truth values.
y_pred	The predicted values.
from_logits	Whether y_pred is expected to be a logits tensor. By default, we assume that y_pred encodes a probability distribution.
axis	Defaults to -1. The dimension along which the entropy is computed.
ignore_class	Optional integer. The ID of a class to be ignored during loss computation. This is useful, for example, in segmentation problems featuring a "void" class (commonly -1 or 255) in segmentation maps. By default (ignore_class=None), all classes are considered.

Returns
Sparse categorical crossentropy loss value.

Figure 18. Network implementation – Cross-entropy.

Multiclass Classification

```
# create model
model = tf.keras.models.Sequential()
model.add(tf.keras.Input(shape=(2,)))
model.add(tf.keras.layers.Dense(8, activation=tf.keras.activations.relu))
model.add(tf.keras.layers.Dense(4, activation=tf.keras.activations.softmax))

# training
optimizer = tf.keras.optimizers.SGD(lr=0.1)
model.compile(loss=tf.keras.metrics.sparse_categorical_crossentropy,
optimizer=optimizer)
model.fit(X, y, batch_size=32, epochs=512)
```

Knowledge Check 3

You are going to train a neural network to decide whether a coin is showing head or tail. To do so, you took many pictures of different coins and manually annotated which side is visible in each picture. The color of the pixels in a picture will serve as input to the network, and the output should somehow specify if the coin depicted in the input is showing head or tails. Which final activation function and loss function should be used to train this neural network?

A None & MSE

B Sigmoid & Binary Cross-entropy

C Softmax & Categorical Cross-entropy

You have reached the end of the lecture.

Image/Figure References

Figure 1. Deep learning APIs.
Figure 2. Logo for Keras deep learning API. Figure 3. Network implementation - Input. Source: https://www.tensorflow.org/api_docs/python/tf/keras/Input
Figure 4. Network implementation – Dense layer. Source: https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dense
Figure 5. Network implementation – ReLU. Source: https://www.tensorflow.org/api_docs/python/tf/keras/activations/relu
Figure 6. Network implementation – Sequential. Source: https://www.tensorflow.org/api_docs/python/tf/keras/Sequential
Figure 7. Network implementation – Model. Source: https://www.tensorflow.org/api_docs/python/tf/keras/Model
Figure 8. Training – MSE loss. Source: https://www.tensorflow.org/api_docs/python/tf/keras/losses/MSE
Figure 9. Training – SGD optimizer. Source: https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/SGD
Figure 10. Training – Model compile. Source: https://www.tensorflow.org/api_docs/python/tf/keras/Model#compile
Figure 11. Training – Model fit. Source: https://www.tensorflow.org/api_docs/python/tf/keras/Model#fit
Table 1. Final activation and loss functions.
Figure 12. Binary classification.
Figure 13. Binary classification.
Figure 14. Network implementation – Sigmoid. Source: https://www.tensorflow.org/api_docs/python/tf/keras/activations/sigmoid
Figure 15. Network implementation – Binary cross-entropy. Source: https://www.tensorflow.org/api_docs/python/tf/keras/metrics/binary_crossentropy
Figure 16. Multiclass classification..
Figure 17. Network implementation – Softmax. Source: https://www.tensorflow.org/api_docs/python/tf/keras/activations/softmax
Figure 18. Network implementation – Cross-entropy. Source: https://www.tensorflow.org/api_docs/python/tf/keras/metrics/sparse_categorical_crossentropy