Introduction to Keras

Keras
https://keras.io

Keras: Deep Learning for humans

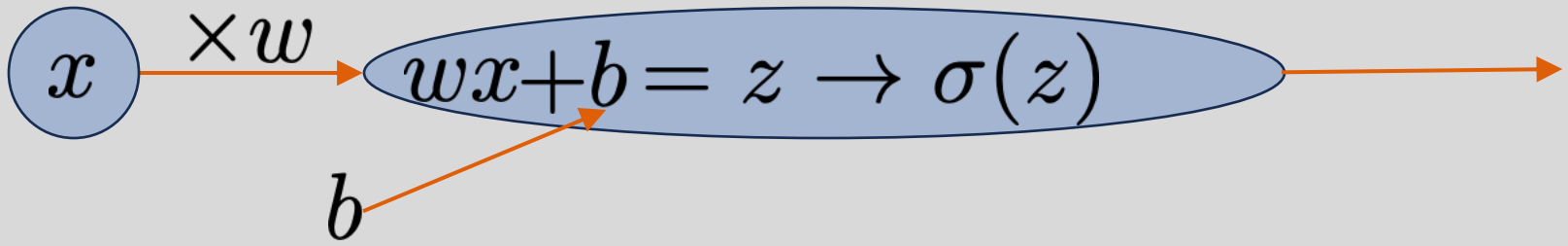
From Lecture 10

Artificial Neural Networks **(ANNs)** are core components for ***deep learning***

Their power is backed by ***universal approximation theorem***(s), which says something about how ANNs can effectively approximate functions of arbitrary complexity (in principle)

We can have some intuition for why UAT works by considering 2-neuron perceptron to locally approximate a function

ANNs feature input layers, hidden layers, and output layers; the layers contain "neurons" that perform straightforward mathematical operations



The Landscape of Machine Learning APIs

Keras:
a High-level API primarily built on top of TensorFlow

Advantages:

•**Ease of use**: Simple, readable syntax makes it beginner-friendly. You can quickly define models without worrying about low-level details.
•**Fast prototyping**: Keras is designed to make experimentation with models faster.
•**TensorFlow integration**: Seamlessly integrates with TensorFlow, offering access to TensorFlow's ecosystem
•**Pre-built models**: Extensive collection of pre-trained models for image classification, text, and time series.

Disadvantages:

•**Limited flexibility**: It is not as flexible or granular as other frameworks like PyTorch when it comes to building custom models or fine-tuning specific layers and operations.
•**Less control**: If you need low-level debugging or intricate control over model training, Keras might feel restrictive.

PyTorch:
a *Standalone framework with dynamic computation*

Advantages:

•**Usage**: Ideal for research and complex, custom model development
•**Flexibility**: PyTorch offers more control over every aspect of model building. Its dynamic computational graph allows for on-the-fly changes
•**Eager execution**: This feature simplifies debugging and experimentation because code can be executed step by step, making it easier to track variable values in real time.
•**Strong GPU support**: Efficiently leverages GPUs for training and integration with NVIDIA libraries (e.g., CUDA)

Disadvantages:

•**Steeper learning curve**: It can be more difficult to grasp for beginners due to its lower-level nature.
•**Less straightforward for fast prototyping**: Compared to Keras, it requires more code to build models

Alternatives

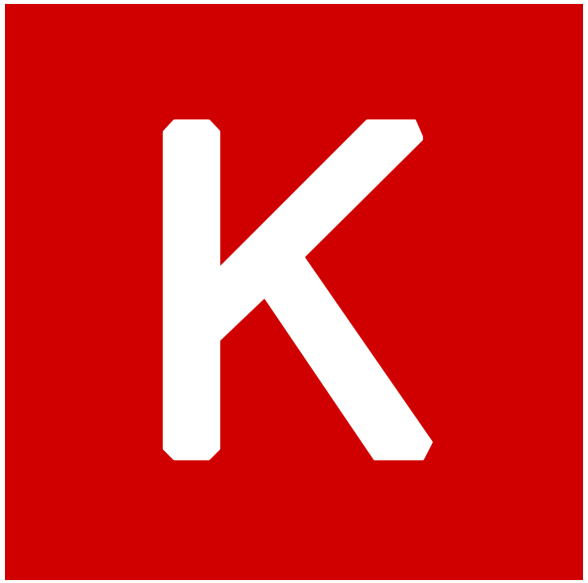
TensorFlow: Core deep learning framework with Keras API. Ideal for those needing granular control, scalability, and production deployment. Optimized for large-scale distributed training but has a steeper learning curve.

JAX: High-performance machine learning library for automatic differentiation and numerical optimization. Excellent for research and complex models, with rapid execution on TPUs/GPUs. Smaller community, steep learning curve.

MXNet: AWS-backed deep learning framework, supporting both symbolic and imperative programming. Scalable for cloud applications, especially AWS. Smaller user base compared to TensorFlow/PyTorch.

FastAI: High-level library built on PyTorch, simplifying deep learning for beginners. Combines PyTorch flexibility with ease of use. Less suited for custom models than raw PyTorch.

The Landscape of Machine Learning APIs



Advantages:

•**Ease of use**: Simple, readable syntax makes it beginner-friendly. You can quickly define models without worrying about low-level details.

•**Fast prototyping**: Keras is designed to make experimentation with models faster.

•**TensorFlow integration**: Seamlessly integrates with TensorFlow, offering access to TensorFlow's ecosystem

•**Pre-built models**: Extensive collection of pre-trained models for image classification, text, and time series.

~~**Disadvantages:**~~

~~•**Limited flexibility**: It is not as flexible or granular as other frameworks like PyTorch when it comes to building custom models or fine-tuning specific layers and operations.~~

~~•**Less control**: If you need low-level debugging or intricate control over model training, Keras might feel restrictive.~~

We are using Keras in this course because…

- it provides a simple, high-level interface that allows students to quickly build and experiment with deep learning models.
- Keras is ideal for beginners, offering an intuitive syntax that helps focus on core machine learning concepts
- It is integrated with TensorFlow, so students will also gain exposure to a production-level framework, which they can build on for more advanced projects in both research and industry.
- You can always transition to PyTorch

The Keras API

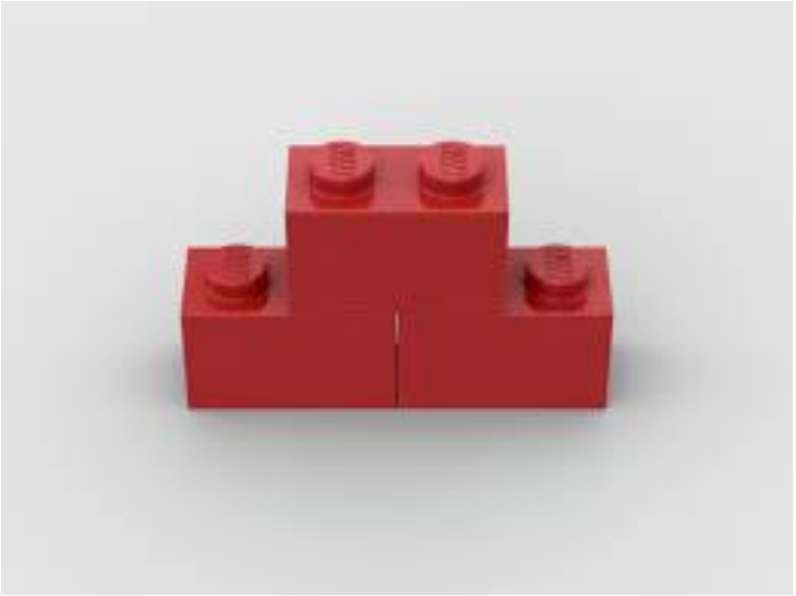
Keras is a deep learning Application Programming Interface that greatly simplifies implementation of machine learning algorithms/math available through TensorFlow

Keras API reference

Models API

- The Model class
- The Sequential class
- Model training APIs
- Model saving & serialization APIs

These are the different containers for creating model objects you start by initializing from here



Layers API

- The base Layer class
- Layer activations
- Layer weight initializers
- Layer weight regularizers
- Layer weight constraints
- Core layers
- Convolution layers
- Pooling layers
- Recurrent layers
- Preprocessing layers
- Normalization layers
- Regularization layers
- Attention layers
- Reshaping layers
- Merging layers
- Locally-connected layers
- Activation layers

These are the building blocks of our models. We connect these different building blocks together to create specific architectures to achieve our modeling objective.

The Basic Steps in Creating Neural Networks

1. You create a model object
2. You add layers to it
3. (You check that everything is OK)
4. You “build” and “compile” the model
5. You train it
6. You test it
7. (You check that everything is OK)
8. You use it
9. (You check that everything is OK)

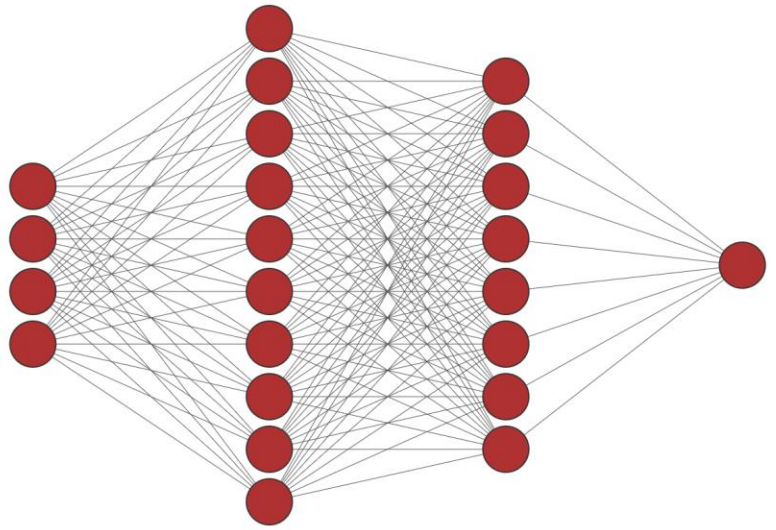
how you do this depends on what API you are using

these are methods of the model object (or member functions of the class)

Preparing a Simple Sequential Model

The Sequential Model API is good enough for preparing simple dense, deep neural networks

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

# Create Model Container
model = keras.Sequential(name="myFirstModel")

# Define Layers
layer1= layers.Dense(10,activation='relu',name="myFirstLayer")
layer2= layers.Dense(8,activation='tanh',name="oldNewsLayer")
output= layers.Dense(1,activation=None,name="output")

# Add layers to model
model.add(keras.Input(shape=(4,)))
model.add(layer1)
model.add(layer2)
model.add(output)

# Admire Model
model.summary()
```

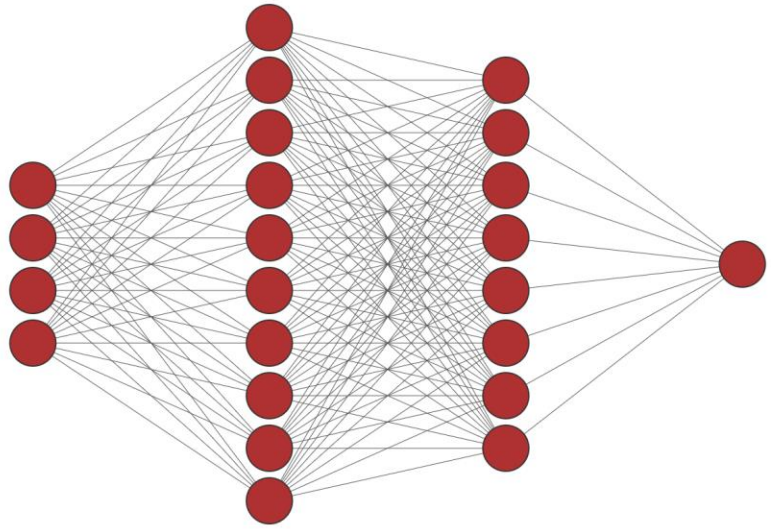


```
Model: "myFirstModel"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
myFirstLayer (Dense)         (None, 10)                50
_________________________________________________________________
oldNewsLayer (Dense)         (None, 8)                 88
_________________________________________________________________
outputLayer (Dense)          (None, 1)                 9
=================================================================
Total params: 147
Trainable params: 147
Non-trainable params: 0
_________________________________________________________________
```

Preparing a Simple Model with Functional API

The Functional Model API is good for simple and complex networks

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

# Define Layers
inputLayer = keras.Input(shape=(4,))
layer1= layers.Dense(10,activation='relu',name="myFirstLayer")
layer2= layers.Dense(8,activation='tanh',name="oldNewsLayer")
output= layers.Dense(1,activation=None,name="output")

# Connect layers using "layer calls"
# we want to achieve
# inputLayer --> layer1 --> layer2 --> outputs
x = layer1(inputLayer)
x = layer2(x)
outputs = output(x)

# Build model from inputs/outputs
model = keras.Model(inputs=inputLayer,outputs=outputs,\
        name="mySecondModel")

# Admire Model
model.summary()
```

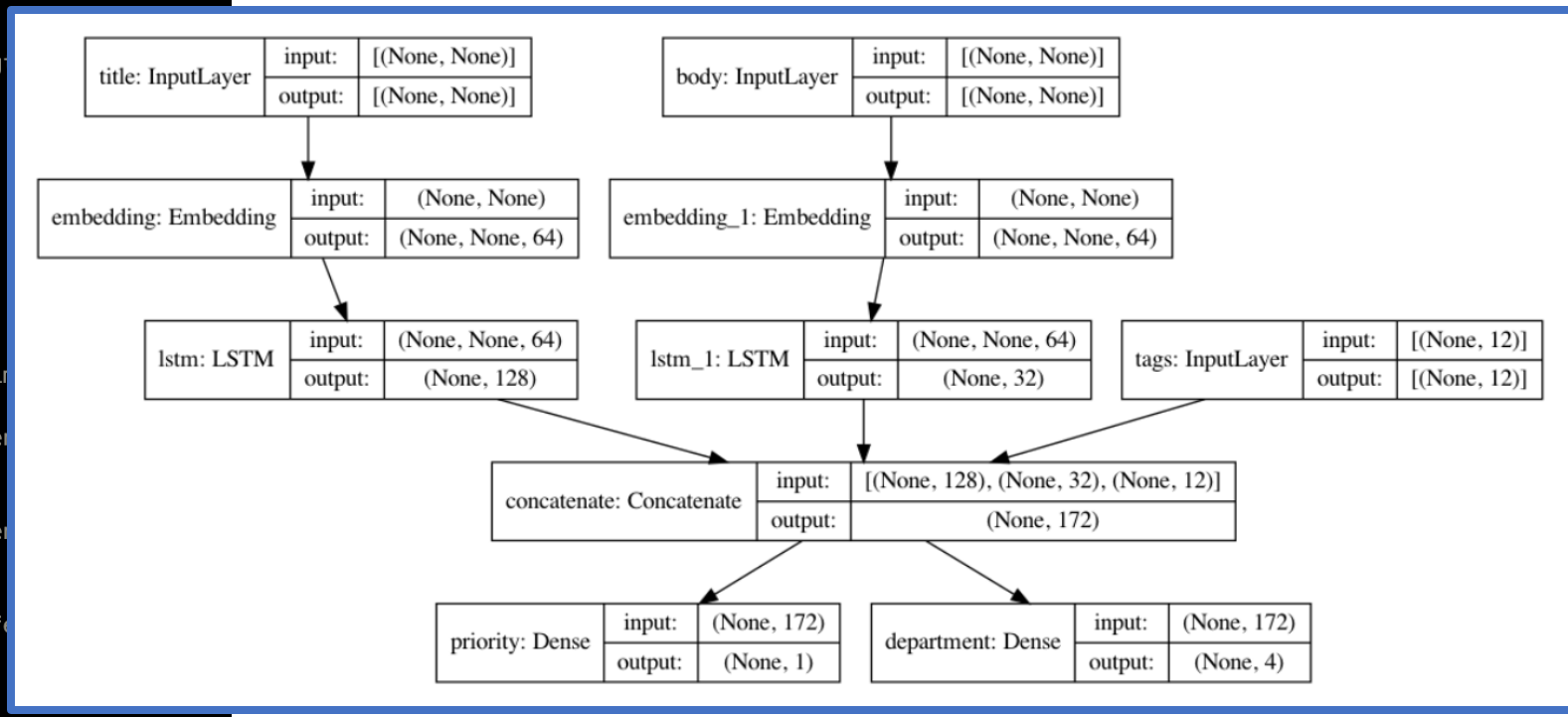


```
Model: "mySecondModel"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_4 (InputLayer)         [(None, 4)]               0
_________________________________________________________________
myFirstLayer (Dense)         (None, 10)                50
_________________________________________________________________
oldNewsLayer (Dense)         (None, 8)                 88
_________________________________________________________________
output (Dense)               (None, 1)                 9
=================================================================
Total params: 147
Trainable params: 147
Non-trainable params: 0
_________________________________________________________________
```

Preparing a Complex Model with Functional API

The Functional Model API is good for simple and complex networks

```
num_tags = 12  # Number of unique issue tags
num_words = 10000  # Size of vocabulary obtained when preprocessing text data
num_departments = 4  # Number of departments for predictions

title_input = keras.Input(
    shape=(None,), name="title"
)  # Variable-length sequence of ints
body_input = keras.Input(shape=(None,), name="body")  # Variable-leng
tags_input = keras.Input(
    shape=(num_tags,), name="tags"
)  # Binary vectors of size `num_tags`

# Embed each word in the title into a 64-dimensional vector
title_features = layers.Embedding(num_words, 64)(title_input)
# Embed each word in the text into a 64-dimensional vector
body_features = layers.Embedding(num_words, 64)(body_input)

# Reduce sequence of embedded words in the title into a single 128-di
title_features = layers.LSTM(128)(title_features)
# Reduce sequence of embedded words in the body into a single 32-dime
body_features = layers.LSTM(32)(body_features)

# Merge all available features into a single large vector via concate
x = layers.concatenate([title_features, body_features, tags_input])

# Stick a logistic regression for priority prediction on top of the f
priority_pred = layers.Dense(1, name="priority")(x)
# Stick a department classifier on top of the features
department_pred = layers.Dense(num_departments, name="department")(x)

# Instantiate an end-to-end model predicting both priority and department
model = keras.Model(
    inputs=[title_input, body_input, tags_input],
    outputs=[priority_pred, department_pred],
)
```



These plots you can make using

`keras.utils.plot_model(modelName, "fileName.png", show_shapes=True)`

Compiling and Training Models

Once you have constructed your desired network architecture, subsequent compilation and training steps are very easy! BUT you should be aware that there are a lot of choices/options

Model training APIs

- compile method
- fit method
- evaluate method
- predict method
- train_on_batch method
- test_on_batch method
- predict_on_batch method
- run_eagerly property

compile method

```
Model.compile(
    optimizer="rmsprop",
    loss=None,
    metrics=None,
    loss_weights=None,
    weighted_metrics=None,
    run_eagerly=None,
    steps_per_execution=None,
    **kwargs
)
```

fit method

```
Model.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,
)
```

By making specifications to these methods, you will affect what your objective for optimization is and how it will be performed; in addition, you may monitor other aspects of training

Optimizers and Loss Metrics

Two specifications that you should have some intuition for are the choice of optimizer and a loss metric

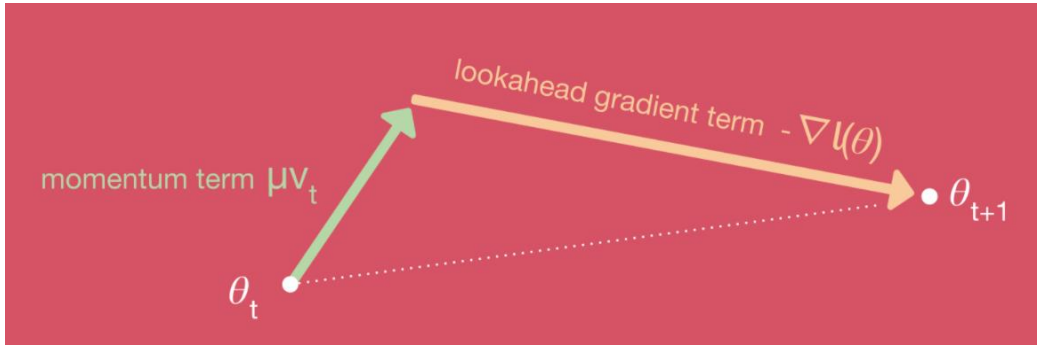
Available optimizers

- SGD
- RMSprop
- Adam
- Adadelta
- Adagrad
- Adamax
- Nadam
- Ftrl

SGD class

Stochastic Gradient Descent (with momentum)

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



Nadam class

Adam optimizer with Nesterov momentum

```
tf.keras.optimizers.Nadam(
    learning_rate=0.001, beta_1=0.9, beta_2=0.999, epsilon=1e-07, name="Nadam", **kwar
)
```

RMSprop class

```
tf.keras.optimizers.RMSprop(
    learning_rate=0.001,
    rho=0.9,
    momentum=0.0,
    epsilon=1e-07,
    centered=False,
    name="RMSprop",
    **kwargs
)
```

root mean square propagation, which uses some moving average of gradients

Optimizers and Loss Metrics

Algorithm 1 Stochastic Gradient Descent

Require: $\alpha_0, \ldots, \alpha_T$: The learning rates for each timestep (presumably annealed)
Require: $f_i(\theta)$: Stochastic objective function parameterized by θ and indexed by timestep i
Require: θ_0: The initial parameters
  while θ_t not converged **do**
    $t \leftarrow t + 1$
    $\mathbf{g}_t \leftarrow \nabla_{\theta_{t-1}} f_t(\theta_{t-1})$
    $\theta_t \leftarrow \theta_{t-1} - \alpha_t \mathbf{g}_t$
  end while
  return θ_t

Algorithm 2 Nesterov-accelerated Adaptive Moment Estimation (Nadam)

Require: $\alpha_0, \ldots, \alpha_T$; $\mu_0, \ldots, \mu_T$; ν; ϵ: Hyperparameters
  $\mathbf{m}_0; \mathbf{n}_0 \leftarrow 0$ (first/second moment vectors)
  while θ_t not converged **do**
    $\mathbf{g}_t \leftarrow \nabla_{\theta_{t-1}} f_t(\theta_{t-1})$
    $\mathbf{m}_t \leftarrow \mu_t \mathbf{m}_{t-1} + (1 - \mu_t)\mathbf{g}_t$
    $\mathbf{n}_t \leftarrow \nu \mathbf{n}_{t-1} + (1 - \nu)\mathbf{g}_t^2$
    $\hat{\mathbf{m}} \leftarrow (\mu_{t+1}\mathbf{m}_t / (1 - \prod_{i=1}^{t+1} \mu_i)) + ((1 - \mu_t)\mathbf{g}_t / (1 - \prod_{i=1}^{t} \mu_i))$
    $\hat{\mathbf{n}} \leftarrow \nu \mathbf{n}_t / (1 - \nu^t)$
    $\theta_t \leftarrow \theta_{t-1} - \frac{\alpha_t}{\sqrt{\hat{\mathbf{n}}_t + \epsilon}} \hat{\mathbf{m}}_t$
  end while
  return θ_t

Optimizers and Loss Metrics

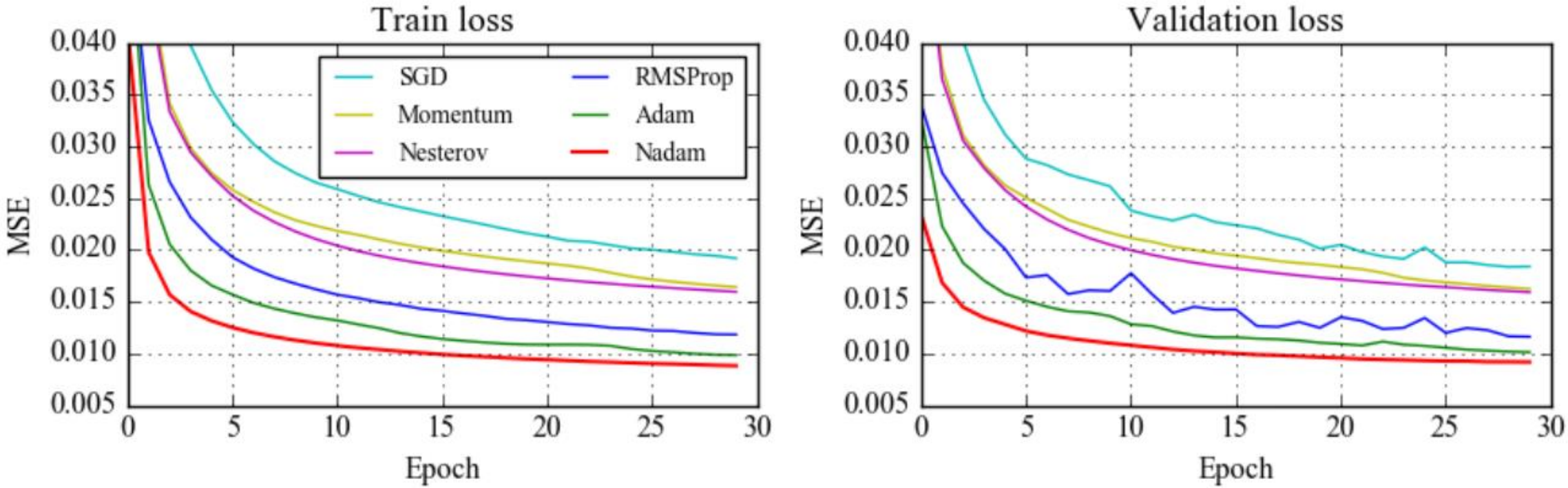


Figure 1: Training and validation loss of different optimizers on the MNIST dataset

Optimizers and Loss Metrics

Two specifications that you should have some intuition for are the choice of optimizer and a loss metric

Probabilistic losses

- BinaryCrossentropy class
- CategoricalCrossentropy class
- SparseCategoricalCrossentropy class
- Poisson class
- binary_crossentropy function
- categorical_crossentropy function
- sparse_categorical_crossentropy function
- poisson function
- KLDivergence class
- kl_divergence function

Regression losses

- MeanSquaredError class
- MeanAbsoluteError class
- MeanAbsolutePercentageError class
- MeanSquaredLogarithmicError class
- CosineSimilarity class
- mean_squared_error function
- mean_absolute_error function
- mean_absolute_percentage_error function
- mean_squared_logarithmic_error function
- cosine_similarity function
- Huber class
- huber function
- LogCosh class
- log_cosh function

`MeanSquaredError` class

```
tf.keras.losses.MeanSquaredError(reduction="auto", name="mean_squared_error")
```

Computes the mean of squares of errors between labels and predictions.

```
loss = square(y_true - y_pred)
```

Usage with the `compile()` API:

```
model.compile(optimizer='sgd', loss=tf.keras.losses.MeanSquaredError())
```

You can also specify just the string name during model compilation

```
model.compile(optimizer="SGD",loss="mean_squared_error",metrics=["mean_absolute_error"])
```

Model Metrics

The loss will indicate what the objective for optimization actually is, but you can also monitor a variety of other quantities related to the model that should be specified at compilation

Accuracy metrics

- Accuracy class
- BinaryAccuracy class
- CategoricalAccuracy class
- SparseCategoricalAccuracy class
- TopKCategoricalAccuracy class
- SparseTopKCategoricalAccuracy class

Probabilistic metrics

- BinaryCrossentropy class
- CategoricalCrossentropy class
- SparseCategoricalCrossentropy class
- KLDivergence class
- Poisson class

Classification metrics based on True/False positives & negatives

- AUC class
- Precision class
- Recall class
- TruePositives class
- TrueNegatives class
- FalsePositives class
- FalseNegatives class
- PrecisionAtRecall class
- SensitivityAtSpecificity class
- SpecificityAtSensitivity class

Regression metrics

- MeanSquaredError class
- RootMeanSquaredError class
- MeanAbsoluteError class
- MeanAbsolutePercentageError class
- MeanSquaredLogarithmicError class
- CosineSimilarity class
- LogCoshError class

Custom Metrics and Losses

The loss will indicate what the objective for optimization actually is, but you can also monitor a variety of other quantities related to the model that should be specified at compilation

```
import tensorflow as tf
```

```
def my_loss_fn(y_true, y_pred):
    squared_difference = tf.square(y_true - y_pred)
    return tf.reduce_mean(squared_difference, axis=-1)  # Note the `axis=-1`

model.compile(optimizer='adam', loss=my_loss_fn)
```

```
def my_metric_fn(y_true, y_pred):
    squared_difference = tf.square(y_true - y_pred)
    return tf.reduce_mean(squared_difference, axis=-1)  # Note the `axis=-1`

model.compile(optimizer='adam', loss='mean_squared_error', metrics=[my_metric_fn])
```

It is straightforward enough to define your own functions for monitoring model performance or for targeting optimization

Model Training

Training a model also has a few options for consideration, but most of the hard work is already done

fit method

```
Model.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,
)
```

Batch size *– this specifies how many examples are used for each gradient update (this defaults to 32; reconsider if you have very little data)*

Epochs *– this is effectively the number of times that you cycle through the training data*

validation split *– this fraction of data will not be used during training and instead will be used to evaluate a validation loss*

validation data *– rather than using validation split you can directly specify some data to monitor the loss on*

callbacks *– these are functions that can be called during training to do special things*

Go to Keras Notebook