CNN Fundamentals & Data Pipeline Best Practices (TensorFlow)

This document summarizes the key practical lessons I learned while implementing a basic CNN image classifier using TensorFlow/Keras.

Although CNN architecture matters, I found that **data pipeline design (splitting, shuffling, batching, caching)** often has a larger impact on performance, stability, and reproducibility.

This note focuses on **engineering correctness**, not just model building.

1. Handling images in a single folder

Problem: Dataset

```
DATA_DIR/

  horse01.jpg

  horse02.jpg

  human01.jpg

  human02.jpg
```

No subfolders → TensorFlow cannot auto-label.

Option A — Parse labels from filenames (manual)

```
files = os.listdir(DATA_DIR)
paths = [os.path.join(DATA_DIR, f) for f in files]
labels = [0 if f.startswith('horse') else 1 for f in files]
dataset = tf.data.Dataset.from_tensor_slices((paths, labels))
```

Option B — Restructure folders (recommended)

```
DATA_DIR/
  horses/
  humans/
```

```
import os
import shutil
```

```
DATA_DIR = “DATA_DIR”
horses_dir = os.path.join(DATA_DIR, "horses")
humans_dir = os.path.join(DATA_DIR, “humans")
# create folders if not exist
os.makedirs(horses_dir, exist_ok=True)
os.makedirs(humans_dir, exist_ok=True)
files = os.listdir(DATA_DIR)
for f in os.listdir(DATA_DIR):
    src = os.path.join(DATA_DIR, f)
    if not os.path.isfile(src):
        continue   # skip folders
    if f.startswith('horse'):
        shutil.move(src, horses_dir)
    elif f.startswith('human'):
        shutil.move(src, humans_dir)
```

Cleaner, scalable, less error-prone.

2. Train / Validation split methods

Method 1 — Manual (`tf.data`)

```
dataset = dataset.shuffle(len(files), seed=42)
train_size = int(0.8 * len(files))
train_ds = dataset.take(train_size)
val_ds   = dataset.skip(train_size)
```

✔ maximum control
✔ works for any dataset format

Method 2 — sklearn

```
from sklearn.model_selection import train_test_split
train_f, val_f, train_l, val_l = train_test_split(
    files, labels,
    test_size=0.2,
    stratify=labels,
    random_state=42
)
```

✔ simple
✔ good for small datasets

Method 3 — TensorFlow utility (easiest)

```
train_ds = tf.keras.utils.image_dataset_from_directory(
    DATA_DIR,
    validation_split=0.2,
    subset="training",
    seed=42
)
val_ds = tf.keras.utils.image_dataset_from_directory(
    DATA_DIR,
    validation_split=0.2,
    subset="validation",
    seed=42
)
```

✔ automatic labels
✔ automatic split
✔ recommended for folder datasets

Note: The functions `tf.keras.preprocessing.image_dataset_from_directory` and `tf.keras.utils.image_dataset_from_directory` are essentially the **same function**, with the latter being the current and recommended path in modern TensorFlow versions. The `tf.keras.preprocessing` module is deprecated.

3. Output layer design

The final layer of a CNN must match the **type of prediction problem**.

Task	Units	Activation
Binary	1	Sigmoid
Multi-class	num_classes	Softmax

Binary example

```
tf.keras.layers.Dense(1, activation='sigmoid')
```

Multi-class example

```
tf.keras.layers.Dense(num_classes, activation=‘softmax')
```

4. Loss & optimizer pairing

Activation and loss functions are mathematically coupled — using the wrong pair leads to unstable training or poor convergence.

Output	Loss
Sigmoid	binary_crossentropy
Softmax	sparse_categorical_crossentropy

Example

```
model.compile(
    optimizer='adam',
    loss='binary_crossentropy',
    metrics=['accuracy']
)
```

Sigmoid

- maps logits → probability in [0,1]
- represents P(y=1∣x)

Binary cross-entropy

- measures distance between predicted probability and true label
- derived from Bernoulli likelihood (maximum likelihood estimation)

$$L = -[y\log(p)+(1-y)\log(1-p)]$$

Key idea

Binary problem → Bernoulli distribution → sigmoid + BCE

Multi-class (single label, >2 classes)

Examples:

- 10-digit classification
- disease type classification

Architecture

```
tf.keras.layers.Dense(num_classes, activation="softmax")
```

Loss

```
sparse_categorical_crossentropy
# or categorical_crossentropy
```

Softmax

- converts logits → probability distribution
- all probabilities sum to 1
- models categorical distribution

Cross-entropy

- measures distance between predicted distribution and true distribution

Sparse vs categorical

Loss	Labels format
sparse_categorical_crossentropy	integers (0,1,2,3)
categorical_crossentropy	one-hot ([0,0,1,0])

Sparse is simpler and more memory efficient.

Key idea

Multi-class → categorical distribution → softmax + CE

When NOT to use tanh

tanh outputs [-1, 1]

Problems:

- not probabilistic
- harder interpretation
- poorer gradient behavior for classification

Typically used for:

- hidden layers
- regression with symmetric outputs

Not recommended for classification outputs.

Task	Output	Activation	Loss
Binary	1	Sigmoid	binary_crossentropy
Multi-class	num_classes	Softmax	sparse_categorical_crossentropy
Regression	1	Linear	Mse

5. Image normalization (rescaling)

Pixel values:

0–255 → too large for NN

Normalize:

0–1

Manual

```
img = img / 255.0
```

One-line pipeline

```
.map(lambda x, y: (x/255.0, y))
```

Recommended (layer)

```
layers.Rescaling(1./255)
```

6. Shuffle BEFORE split

Case 1

```
image_dataset_from_directory(
    shuffle=True,
    batch_size=32
)
```

Internally does:

shuffle(samples) → batch

Effect:

✅ sample-level shuffle
✅ batches formed from randomized samples

This is statistically correct.

Case 2

```
image_dataset_from_directory(
    shuffle=False,
    batch_size=32
)
.cache()
.shuffle()
```

Pipeline:
batch → shuffle(batches)

Effect:

❌ only batches shuffled
❌ samples inside batch fixed

This is weaker randomness.

Case 3

```
image_dataset_from_directory(
    shuffle=False,
)
.cache()
.shuffle()
.batch()
```

Pipeline:
shuffle(samples) → batch

Effect:

✅ sample-level shuffle
✅ statistically identical to Case 1

Even though Case 1 and 3 are equivalent statistically:

Case 3 advantages:

- shuffle happens after cache (faster)
- adjustable buffer size
- clearer control
- easier augmentation insertion
- standard tf.data engineering

So it's about **performance + flexibility**, not correctness.

7. Shuffle buffer size

What does shuffle(buffer_size) actually mean?

It does **NOT** mean:

❌ “shuffle 1000 times”
❌ “shuffle 1000 samples per batch”

It means:

“keep a buffer of N elements in memory and randomly sample from it”

How TensorFlow shuffle works internally

TensorFlow uses a **streaming random buffer algorithm**, not a full permutation.

Algorithm (conceptually)

Given:

```
.shuffle(buffer_size=3)
dataset = [1,2,3,4,5,6,7]
```

Step-by-step:

Fill buffer first

```
buffer = [1,2,3]
```

Then repeat:

1. randomly pick one element from buffer → output it
2. replace it with next element from dataset

Example:

pick 2 → output
buffer becomes [1,3,4]

pick 4 → output
buffer becomes [1,3,5]

pick 1 → output
buffer becomes [6,3,5]
...

Key implication

buffer controls randomness quality

If buffer is SMALL

Example:

```
shuffle(3)
```

You only mix among 3 nearby samples.

Result:

almost ordered
Bad randomness.

If buffer equals dataset size

Example:

```
shuffle(1000)  # dataset has 1000 samples
```

Now:

buffer = entire dataset
So this becomes:

perfect Fisher–Yates shuffle (true random permutation)

Best randomness.

Why people use 1000?

Because many small datasets are around:

- cats vs dogs ≈ 2000
- horses vs humans ≈ 1000

So:

buffer ≈ dataset size
→ near-perfect shuffle.

Rule of thumb

Ideal:

buffer >= dataset size

Good:

buffer >= a few thousand

Bad:

buffer << dataset size

Memory tradeoff

Bigger buffer:

- more RAM
- better randomness

Smaller buffer:

- less RAM
- weaker randomness

For images:

buffer 1000 × (150×150×3 float32) ≈ 250MB

Practical recommendation

small/medium datasets (<10k)

shuffle(1000–5000)

very large datasets (>100k)

shuffle(5000–10000) (no need full size)

tiny datasets (<100)

shuffle(dataset_size)

8. Recommended production-style pipeline

```
train_ds = tf.keras.utils.image_dataset_from_directory(
    DATA_DIR,
    validation_split=0.2,
    subset="training",
    seed=42,
    shuffle=False,
    batch_size=None
)

val_ds = tf.keras.utils.image_dataset_from_directory(
    DATA_DIR,
    validation_split=0.2,
    subset="validation",
    seed=42,
    shuffle=False,
    batch_size=None
)

train_ds = (
    train_ds
    .cache()
    .shuffle(1000)
    .batch(32)
    .prefetch(tf.data.AUTOTUNE)
)
```

```
val_ds = (
    val_ds
    .cache()
    .batch(32)
    .prefetch(tf.data.AUTOTUNE)
)
```

Additional practical tips

✔ Always check class balance

```
train_ds.class_names
```

✔ Start simple CNN first

Deeper ≠ better. Overfitting is common.

✔ Data quality > model complexity

Correct labels + proper split matter more than architecture.

✔ Reproducibility

Always set:

seed=42

Key takeaway

From experience:

20% model design
80% data pipeline correctness
A clean pipeline often improves performance more than adding layers.