Lab 5

In this lab, you'll build and evaluate a simple fully-connected neural network on the Fashion-MNIST dataset, then explore two classic regularization strategies: L_2 weight decay and Dropout to combat overfitting.

Dataset Description

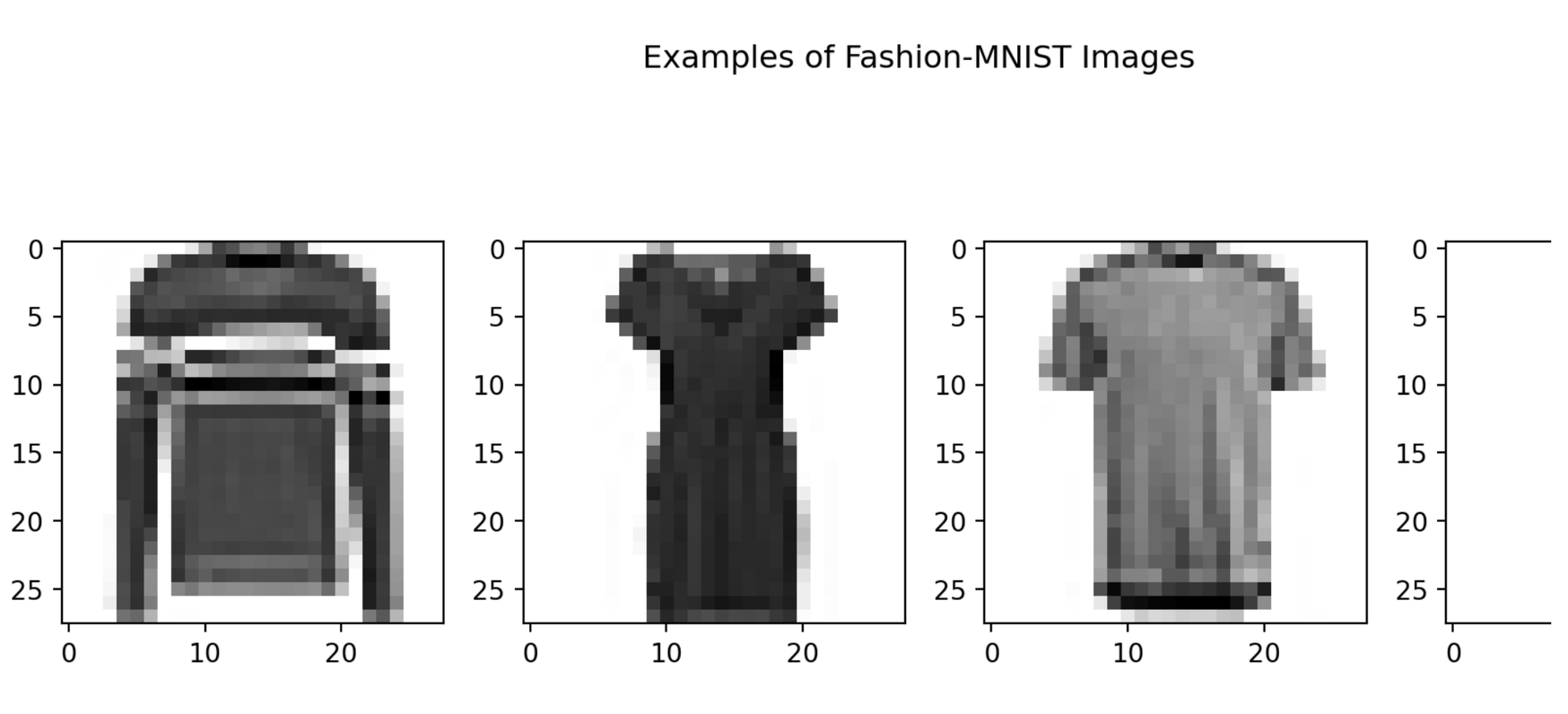
The **Fashion-MNIST** dataset is a drop-in replacement for the original MNIST (handwritten digits) dataset, but contains 70,000 grayscale images of Zalando's article images:

- **Total images**: 70,000 (60,000 training, 10,000 test)
- **Image size**: 28×28 pixels
- **Color**: Grayscale (single channel)
- **Classes**: 10 categories (T-shirt/top, Trouser, Pullover, Dress, Coat, Sandal, Shirt, Sneaker, Bag, Ankle boot)
- **Source**: Provided by Zalando Research, publicly available via GitHub and built into Keras/TensorFlow

Each class is balanced with exactly 6,000 images in the training set and 1,000 in the test set, making it ideal for benchmarking new models.

Load Data & Inspect Examples

```
import tensorflow as tf
from keras import layers, models
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

## Let's Load The data
(fx_train, fy_train), (fx_test, fy_test) = tf.keras.datasets.fashion_mnist.load_data()

## Assign class names
y = fy_train.flatten()

label_map = {
    0: "T-shirt/top", 1: "Trouser",     2: "Pullover",
    3: "Dress",       4: "Coat",        5: "Sandal",
    6: "Shirt",       7: "Sneaker",     8: "Bag",
    9: "Ankle boot"
}

#### Assigning true labels
df = pd.DataFrame({'label': y})
df['class_name']  = df['label'].map(label_map)

indices = [5, 20, 789, 4567]  #indices of examples to plot
fig, axes = plt.subplots(1, len(indices), figsize=(10, 4))
fig.suptitle("Examples of Fashion-MNIST Images")
for ax, idx in zip(axes, indices):
    ax.imshow(fx_train[idx], cmap='binary')

plt.tight_layout()
plt.show()
```



Now we print the annotated true labels:

```
print(df.class_name[indices])
```

```
5          Pullover
20            Dress
789     T-shirt/top
4567        Trouser
Name: class_name, dtype: object
```

Let's see if the classes are balanced:

```
### Explore Classes distribution
summary = (
    df['label']
      .value_counts(sort=False)
      .rename_axis('label')
      .reset_index(name='count')
)
summary['percent']     = 100 * summary['count'] / summary['count'].sum()
summary['class_name']  = summary['label'].map(label_map)

# Reorder and drop count
summary = summary[['label', 'class_name', 'percent']]
print(summary)
```

```
   label   class_name  percent
0      9   Ankle boot     10.0
1      0  T-shirt/top     10.0
2      3        Dress     10.0
3      2     Pullover     10.0
4      7      Sneaker     10.0
5      5       Sandal     10.0
6      1      Trouser     10.0
7      6        Shirt     10.0
8      4         Coat     10.0
9      8          Bag     10.0
```

Now we preprocess the data and build the Neural Network architecture:

```
#### Rehsape images into vectors
x_train = fx_train.reshape(-1, 28*28) / 255.0
x_test  = fx_test.reshape(-1, 28*28)  / 255.0

###Build model
model = models.Sequential([
    layers.Input(shape=(28*28,)),
    layers.Dense(128, activation='relu'),
    layers.Dense(64, activation='relu'),
    layers.Dense(10, activation='softmax')])

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

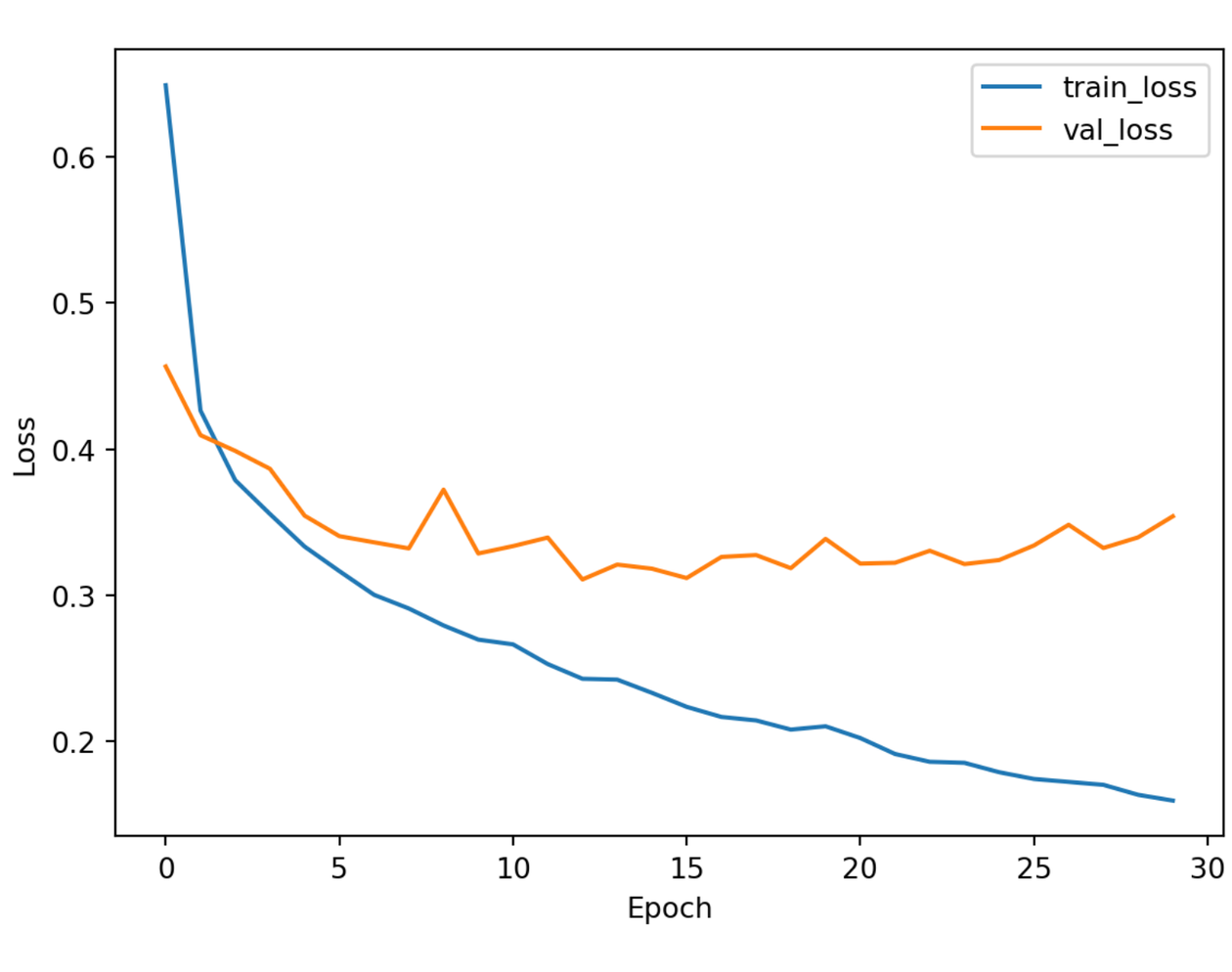
In this next step we actually train our feed-forward network on the Fashion-MNIST training data and simultaneously hold out 20% of it for validation. Here's what happens:

```
###Training the model
history = model.fit(
    x_train, fy_train,
    validation_split=0.2, ##20% for validation
    epochs=30,
    batch_size=256
)
```

```
Epoch 1/30
  1/188 ━━━━━━━━━━━━━━━━━━━━ 52s 280ms/step - accuracy: 0.0938 - loss: 2.3740  2/188 ━━
Epoch 2/30
  1/188 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8906 - loss: 0.3391
Epoch 3/30
  1/188 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8398 - loss: 0.4206
Epoch 4/30
  1/188 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8594 - loss: 0.3465
Epoch 5/30
  1/188 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8867 - loss: 0.3044
Epoch 6/30
  1/188 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8672 - loss: 0.3207
Epoch 7/30
  1/188 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9023 - loss: 0.2878
Epoch 8/30
  1/188 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8789 - loss: 0.3274
Epoch 9/30
  1/188 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8867 - loss: 0.3422
Epoch 10/30
  1/188 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8594 - loss: 0.3454
Epoch 11/30
  1/188 ━━━━━━━━━━━━━━━━━━━━ 1s 9ms/step - accuracy: 0.9102 - loss: 0.2638
Epoch 12/30
  1/188 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8867 - loss: 0.2942
Epoch 13/30
  1/188 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8789 - loss: 0.2594
Epoch 14/30
  1/188 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9414 - loss: 0.1662
Epoch 15/30
  1/188 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9141 - loss: 0.2187
Epoch 16/30
  1/188 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9375 - loss: 0.2000
Epoch 17/30
  1/188 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9414 - loss: 0.1658
Epoch 18/30
  1/188 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9297 - loss: 0.2142
Epoch 19/30
  1/188 ━━━━━━━━━━━━━━━━━━━━ 2s 12ms/step - accuracy: 0.8828 - loss: 0.3181
Epoch 20/30
  1/188 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9141 - loss: 0.1959
Epoch 21/30
  1/188 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9023 - loss: 0.3033
Epoch 22/30
  1/188 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9414 - loss: 0.1743
Epoch 23/30
  1/188 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9375 - loss: 0.1654
Epoch 24/30
  1/188 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9297 - loss: 0.2010
Epoch 25/30
  1/188 ━━━━━━━━━━━━━━━━━━━━ 2s 13ms/step - accuracy: 0.9414 - loss: 0.1316
Epoch 26/30
  1/188 ━━━━━━━━━━━━━━━━━━━━ 2s 11ms/step - accuracy: 0.9414 - loss: 0.1654
Epoch 27/30
  1/188 ━━━━━━━━━━━━━━━━━━━━ 2s 12ms/step - accuracy: 0.9375 - loss: 0.1662
Epoch 28/30
  1/188 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9297 - loss: 0.2086
Epoch 29/30
  1/188 ━━━━━━━━━━━━━━━━━━━━ 1s 11ms/step - accuracy: 0.9141 - loss: 0.1832
Epoch 30/30
  1/188 ━━━━━━━━━━━━━━━━━━━━ 1s 11ms/step - accuracy: 0.9375 - loss: 0.1889
```

We can plot the training vs.validation loss curves with:

```
####Plot training Loss and validation loss over epochs
plt.figure()
plt.plot(history.history['loss'], label='train_loss')
plt.plot(history.history['val_loss'], label='val_loss')
plt.xlabel('Epoch'); plt.ylabel('Loss'); plt.legend()
plt.show()
```



This visualization lets us see how quickly the model is learning (the downward slope of the training loss), and how the validation loss eventually starts rising (a classic sign of over-fitting once the network has learned too much noise).

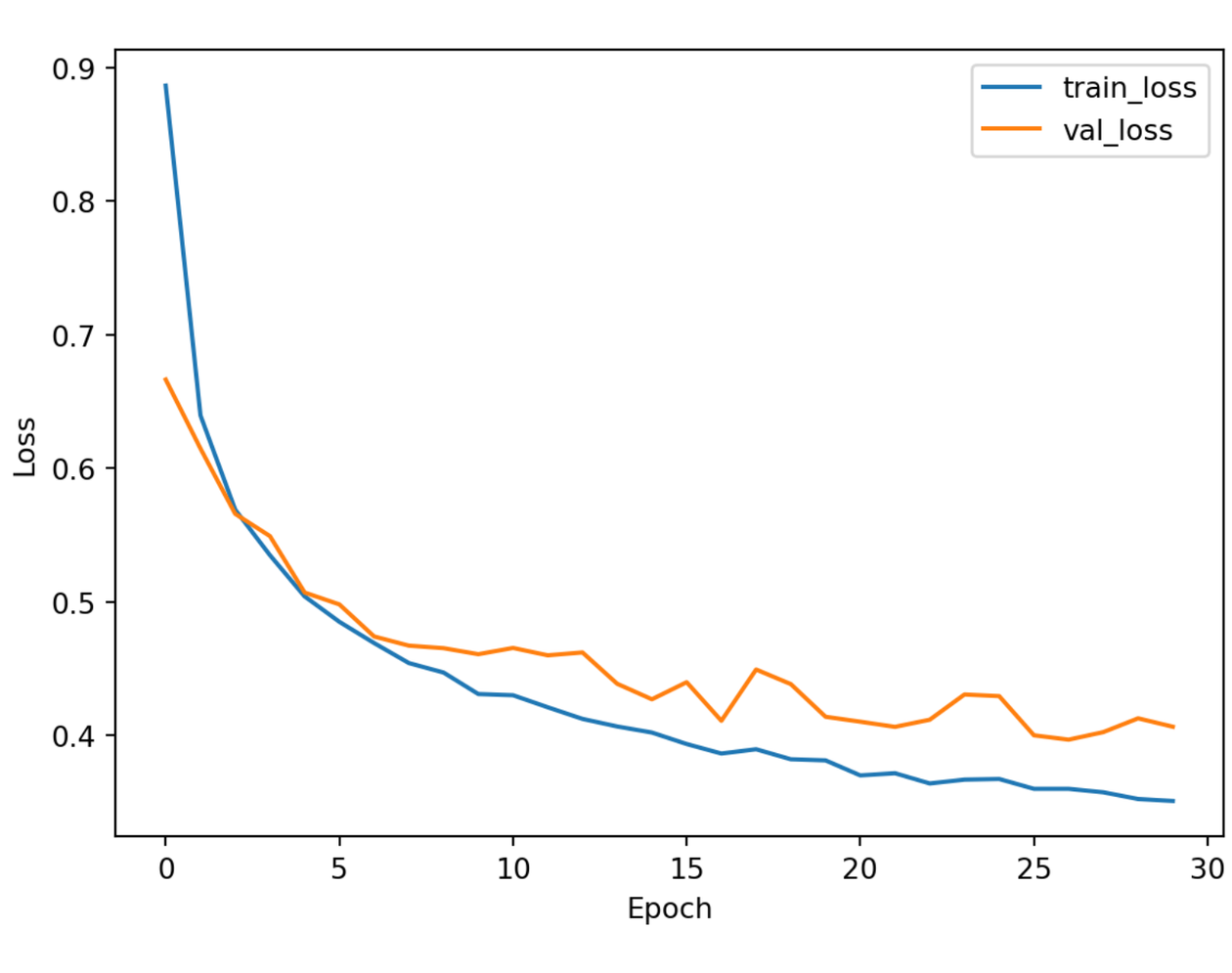
Fixing Overfitting

L2 Weight Regularization

Penalize large weights by adding an l_2 penalty to the loss:

```
from keras.regularizers import l2
model_l2 = models.Sequential([
    layers.Input(shape=(28*28,)),
    layers.Dense(128, activation='relu',
                 kernel_regularizer=l2(1e-3)), ## Penalty coefficient lambda = 1e-3
    layers.Dense(64, activation='relu',
                 kernel_regularizer=l2(1e-3)),
    layers.Dense(10, activation='softmax')
])

model_l2.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

history = model_l2.fit(
    x_train, fy_train,
    validation_split=0.2, ##20% for validation
    epochs=30,
    batch_size=256
)

plt.figure()
plt.plot(history.history['loss'], label='train_loss')
plt.plot(history.history['val_loss'], label='val_loss')
plt.xlabel('Epoch'); plt.ylabel('Loss'); plt.legend()
plt.show()
```

```
Epoch 1/30
  1/188 ━━━━━━━━━━━━━━━━━━━━ 53s 287ms/step - accuracy: 0.1328 - loss: 2.6365
Epoch 2/30
  1/188 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8594 - loss: 0.6269
Epoch 3/30
  1/188 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8633 - loss: 0.5588
Epoch 4/30
  1/188 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8789 - loss: 0.4843
Epoch 5/30
  1/188 ━━━━━━━━━━━━━━━━━━━━ 2s 12ms/step - accuracy: 0.8203 - loss: 0.5702
Epoch 6/30
  1/188 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8906 - loss: 0.4282
Epoch 7/30
  1/188 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8828 - loss: 0.4926
Epoch 8/30
  1/188 ━━━━━━━━━━━━━━━━━━━━ 2s 11ms/step - accuracy: 0.9023 - loss: 0.4180
Epoch 9/30
  1/188 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8789 - loss: 0.4478
Epoch 10/30
  1/188 ━━━━━━━━━━━━━━━━━━━━ 2s 11ms/step - accuracy: 0.8945 - loss: 0.3872
Epoch 11/30
  1/188 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8750 - loss: 0.5070
Epoch 12/30
  1/188 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9180 - loss: 0.3847
Epoch 13/30
  1/188 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8789 - loss: 0.4605
Epoch 14/30
  1/188 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8945 - loss: 0.4516
Epoch 15/30
  1/188 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9219 - loss: 0.3771
Epoch 16/30
  1/188 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8789 - loss: 0.3818
Epoch 17/30
  1/188 ━━━━━━━━━━━━━━━━━━━━ 1s 11ms/step - accuracy: 0.8906 - loss: 0.3792
Epoch 18/30
  1/188 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8906 - loss: 0.3929
Epoch 19/30
  1/188 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8984 - loss: 0.3800
Epoch 20/30
  1/188 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9023 - loss: 0.3679
Epoch 21/30
  1/188 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9062 - loss: 0.3746
Epoch 22/30
  1/188 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9336 - loss: 0.3046
Epoch 23/30
  1/188 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8789 - loss: 0.3749
Epoch 24/30
  1/188 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8984 - loss: 0.3948
Epoch 25/30
  1/188 ━━━━━━━━━━━━━━━━━━━━ 1s 11ms/step - accuracy: 0.8984 - loss: 0.3358
Epoch 26/30
  1/188 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8594 - loss: 0.4780
Epoch 27/30
  1/188 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9219 - loss: 0.3178
Epoch 28/30
  1/188 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.8984 - loss: 0.3601
Epoch 29/30
  1/188 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9141 - loss: 0.3355
Epoch 30/30
  1/188 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.9180 - loss: 0.3272
```

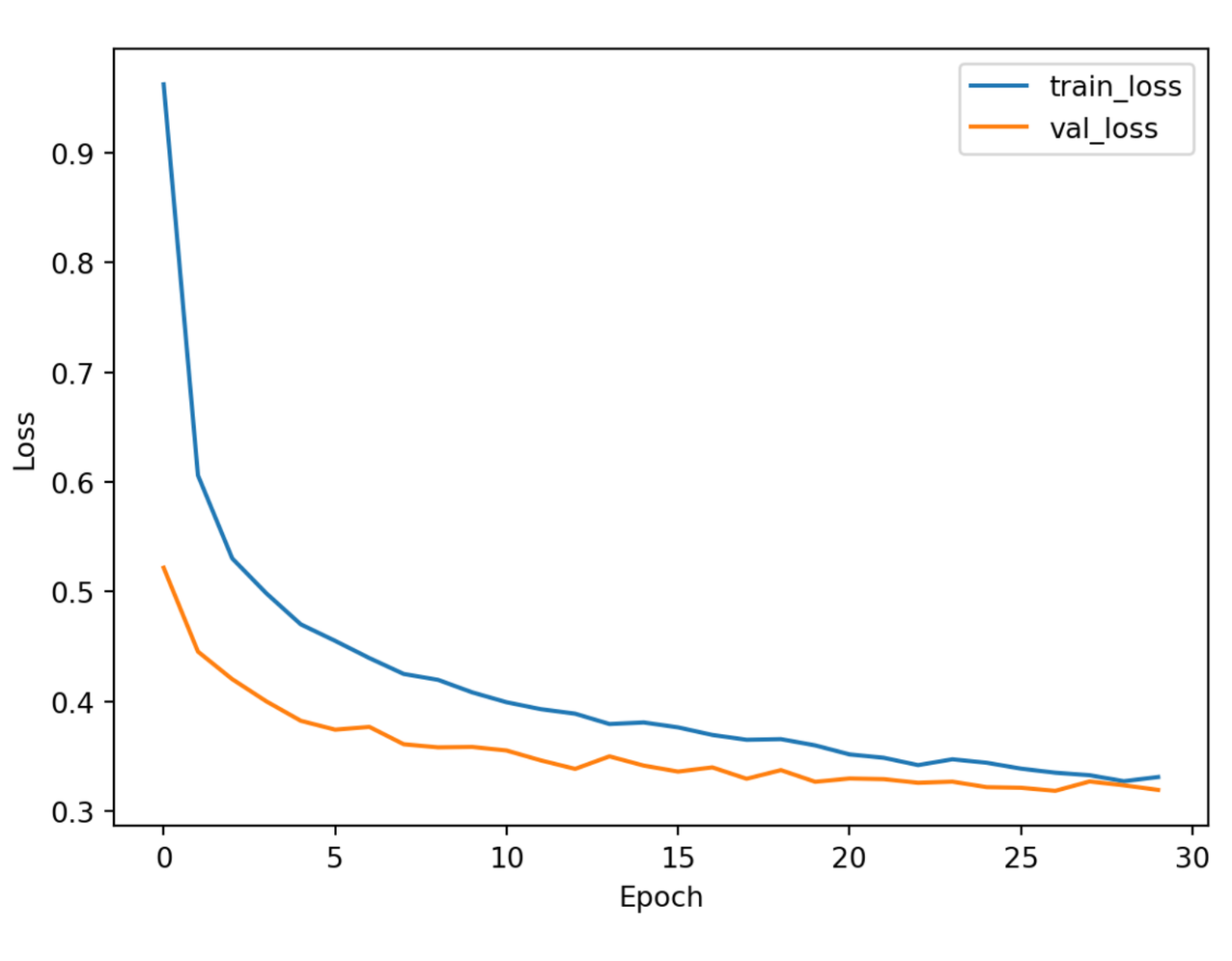


Dropout

Randomly "drop" a fraction of neurons each update so the network can't rely on any one feature:

```
model_do = models.Sequential([
    layers.Input(shape=(28*28,)),
    layers.Dense(128, activation='relu'),
    layers.Dropout(0.5),              #drop 50% of activations
    layers.Dense(64, activation='relu'),
    layers.Dropout(0.3),              #drop 30% activations
    layers.Dense(10, activation='softmax')
])

model_do.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

history = model_do.fit(
    x_train, fy_train,
    validation_split=0.2, ##20% for validation
    epochs=30,
    batch_size=256
)

plt.figure()
plt.plot(history.history['loss'], label='train_loss')
plt.plot(history.history['val_loss'], label='val_loss')
plt.xlabel('Epoch'); plt.ylabel('Loss'); plt.legend()
plt.show()
```

```
racy: 0.3271 - loss: 1.9450

0.7666 - loss: 0.6689

0.8117 - loss: 0.5426

0.8235 - loss: 0.5160

0.8354 - loss: 0.4760

0.8256 - loss: 0.4796

0.8430 - loss: 0.4284

0.8542 - loss: 0.4027

0.8541 - loss: 0.4212

0.8520 - loss: 0.4118

0.8498 - loss: 0.4094

0.8506 - loss: 0.4046

0.8659 - loss: 0.3788

0.8605 - loss: 0.3873

0.8670 - loss: 0.3793

0.8643 - loss: 0.3667

0.8593 - loss: 0.3901

0.8783 - loss: 0.3397

0.8733 - loss: 0.3551

0.8632 - loss: 0.3716

0.8671 - loss: 0.3582

0.8763 - loss: 0.3477

0.8769 - loss: 0.3251

0.8702 - loss: 0.3483

0.8719 - loss: 0.3489

0.8792 - loss: 0.3368

0.8744 - loss: 0.3374

0.8730 - loss: 0.3486

0.8798 - loss: 0.3330

0.8838 - loss: 0.3286
```



Final Evaluation

Evaluate your all three models on the held-out test set:

```
for name, m in [('Baseline', model),
                ('L2',       model_l2),
                ('Dropout',  model_do)]:
    loss, acc = m.evaluate(x_test, fy_test, verbose=0)
    print(f"{name} Test Accuracy: {acc:.4f}")
```

```
Baseline Test Accuracy: 0.8835
L2 Test Accuracy: 0.8730
Dropout Test Accuracy: 0.8803
```

Extensions & Questions

1. **Combine** both L_2 penalty and Dropout in a single model. Does it outperform either alone?
2. **Hyperparameter tuning:**
   - Sweep $\lambda \in \{10^{-1}, 10^{-2}, 10^{-3}\}$ for L_2
   - Vary Dropout $rates \in \{0.2, 0.5, 0.7\}$
   - Try different batch sizes (64, 128, 256) and optimizers (SGD, Adam, RMSprop)
   - Increase the number of epochs
3. **Model depth:** Increase or decrease the number of hidden layers/units. How does capacity affect overfitting?
4. **Visualize misclassifications:** Plot some examples the model got wrong—what patterns do you observe?