Deep Learning - Classication , K-Fold and Regularization

by Sanjeev Gupta

Contents:

1. Binary Classification Example

First, lets look at an examplary problem of binary classification.

First, let's import all necessary libraries. Note that we importing the [imdb dataset](#) from keras.

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from keras.callbacks import TensorBoard, ModelCheckpoint
from tensorflow.keras.datasets import imdb

import numpy as np
import matplotlib.pyplot as plt
```

Now, let's define a preprocessing function, that we will shortly use.

```
def vectorize_sequences(sequences, dimension=10000):
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        for j in sequence:
            results[i, j] = 1.
    return results

def find_smallest_review(train_data):
  min_l = 100000
  min_i = -1
  for i,sample in enumerate(train_data):
    l = len(sample)
    if l < min_l:
      min_l = l
      min_i = i
  return min_i, min_l
```

Let's load the imdb dataset and viusalize the data before pre-processing it.

```
# Load Data
(train_data, train_labels), (test_data, test_labels) = imdb.load_data(num_words=10000) # num_words is the vocabulary size
print(f"train_data[0] = {train_data[0]}")   # encoded review
print(f"len(train_data[0] ) = {len(train_data[0])}")
print(f"train_labels[0] ={train_labels[0]}")
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/imdb.npz
17464789/17464789 [==============================] - 0s 0us/step
train_data[0] = [1, 14, 22, 16, 43, 530, 973, 1622, 1385, 65, 458, 4468, 66, 3941, 4, 173, 36, 256, 5, 25, 100, 43, 838, 112, 50, 67
len(train_data[0] ) = 218
train_labels[0] =1
```

```
word_index = imdb.get_word_index()
i = 0
for k,v in word_index.items():
  i+=1
  print(k,v)
  if i == 15:
    break
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/imdb_word_index.json
1641221/1641221 [==============================] - 0s 0us/step
fawn 34701
tsukino 52006
nunnery 52007
sonja 16816
vani 63951
woods 1408
spiders 16115
hanging 2345
woody 2289
trawling 52008
hold's 52009
comically 11307
localized 40830
disobeying 30568
'royale 52010
```

```
# Visualize Data
# HW: Go through this cell and understand how it works
sample_i = 0  # visualize 0th sample
word_index = imdb.get_word_index()
reverse_word_index = dict([(value, key) for (key, value) in word_index.items()])
decoded_review = " ".join([reverse_word_index.get(i - 3, "?") for i in train_data[sample_i]])
print(decoded_review)
print(train_labels[sample_i])
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/imdb_word_index.json
1641221/1641221 [==============================] - 0s 0us/step
? this film was just brilliant casting location scenery story direction everyone's really suited the part they played and you could
1
```

```
min_i, min_len = find_smallest_review(train_data)

print(f"train_data[min_i] = {train_data[min_i]}")   # encoded review
print(f"len(train_data[min_i] ) = {len(train_data[min_i])}")
print(f"train_labels[min_i] ={train_labels[min_i]}")
```

```
train_data[min_i] = [1, 13, 586, 851, 14, 31, 60, 23, 2863, 2364, 314]
len(train_data[min_i] ) = 11
train_labels[min_i] =0
```

```
# Visualize Data
sample_i = min_i  # visualize 0th sample
word_index = imdb.get_word_index()
reverse_word_index = dict([(value, key) for (key, value) in word_index.items()])
decoded_review = " ".join([reverse_word_index.get(i - 3, "?") for i in train_data[sample_i]])
print(decoded_review)
print(train_labels[sample_i])
```

```
? i wouldn't rent this one even on dollar rental night
0
```

```
# Preprocess data
x_train = vectorize_sequences(train_data)
x_test = vectorize_sequences(test_data)
y_train = np.asarray(train_labels).astype("float32")
y_test = np.asarray(test_labels).astype("float32")
print(f"x_train.shape ={x_train.shape}")
print(f"y_train.shape ={y_train.shape}")
print(f"x_train[min_i] = {x_train[min_i]}")
print(f"y_train[min_i] ={y_train[min_i]}")

# Q: What is the size of one input tensor?

# Split training set to validation and train sets - simple holdout validation
x_val = x_train[:10000]
partial_x_train = x_train[10000:]
y_val = y_train[:10000]
partial_y_train = y_train[10000:]
```

```
x_train.shape =(25000, 10000)
y_train.shape =(25000,)
x_train[min_i] = [0. 1. 0. ... 0. 0. 0.]
y_train[min_i] =0.0
```

```
print(f"first 15 elements of one-hot code of smallest review= {x_train[min_i][0:15]}")
np.where(x_train[min_i] == 1)  # which indices in the the one-hot code are 1s
```

```
first 15 elements of one-hot code of smallest review= [0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1.]
(array([   1,    13,    14,    23,    31,    60,   314,   586,   851, 2364, 2863]),)
```

This way of splitting the training set into a training and validation set is called **simple holdout validation splitting.**

It is also good practice the shuffle data before the split. We have not done it here.

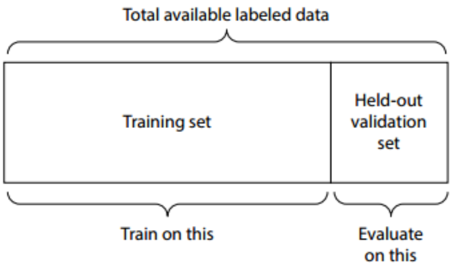


Figure 5.12 Simple holdout validation split

Now that we the data, let's follow the standard steps-

- define a model,
- define the callbacks,
- compile the model,
- fit the model,
- evaluate the model

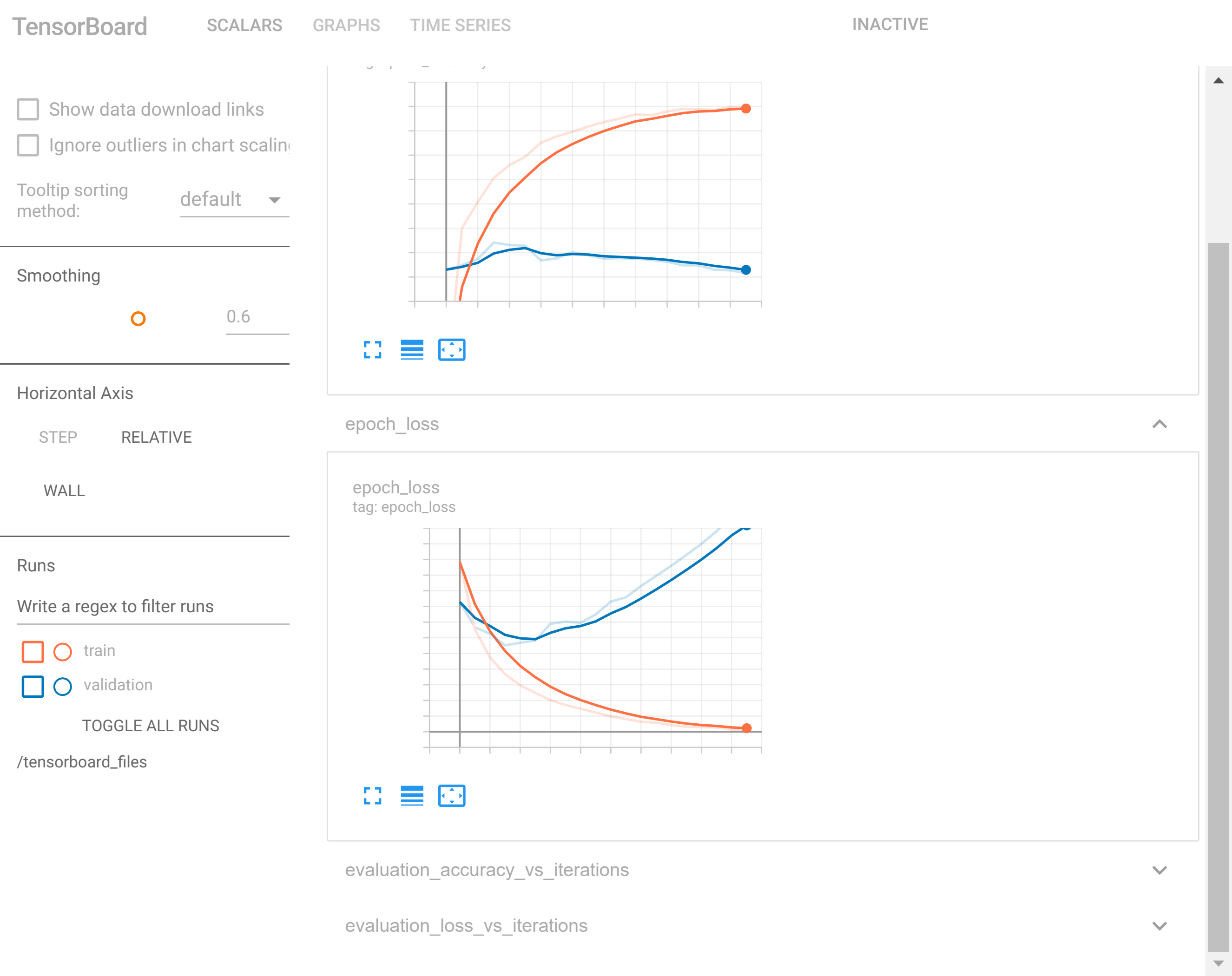
```
# Define a simple model
model = keras.Sequential([
layers.Dense(16, activation="relu"),
layers.Dense(16, activation="relu"),
layers.Dense(1, activation="sigmoid")  # Q: Why only 1 neuron in the output layer?
])


# Define callbacks
callbacks = [ ModelCheckpoint("imdb_model_checkpoint",save_best_only=True),
              TensorBoard(log_dir="/tensorboard_files")]


# Compile the model
model.compile(optimizer="RMSprop",
                loss="binary_crossentropy",
                metrics=["accuracy"])

# Optional HW Exercise: Play with learning rate, momentum and nestorov
# model.compile(optimizer=tf.keras.optimizers.SGD(learning_rate=0.01, momentum=0.9, nesterov=True),
#                 loss="binary_crossentropy",
#                 metrics=["accuracy"])


# Fit the model
history = model.fit(partial_x_train,
                    partial_y_train,
                    epochs=20,
                    batch_size=512,
                    validation_data=(x_val, y_val),
                    callbacks=callbacks)
```

```
Epoch 1/20
30/30 [==============================] - 4s 91ms/step - loss: 0.5680 - accuracy: 0.7675 - val_loss: 0.4481 - val_accuracy: 0.8547
Epoch 2/20
30/30 [==============================] - 2s 72ms/step - loss: 0.3641 - accuracy: 0.8946 - val_loss: 0.3394 - val_accuracy: 0.8841
Epoch 3/20
30/30 [==============================] - 2s 54ms/step - loss: 0.2628 - accuracy: 0.9211 - val_loss: 0.2994 - val_accuracy: 0.8865
Epoch 4/20
30/30 [==============================] - 2s 55ms/step - loss: 0.2038 - accuracy: 0.9390 - val_loss: 0.2781 - val_accuracy: 0.8902
Epoch 5/20
30/30 [==============================] - 1s 36ms/step - loss: 0.1625 - accuracy: 0.9511 - val_loss: 0.2831 - val_accuracy: 0.8876
Epoch 6/20
```

```
30/30 [==============================] - 1s 34ms/step - loss: 0.1344 - accuracy: 0.9605 - val_loss: 0.2828 - val_accuracy: 0.8896
Epoch 7/20
30/30 [==============================] - 1s 39ms/step - loss: 0.1115 - accuracy: 0.9681 - val_loss: 0.2923 - val_accuracy: 0.8879
Epoch 8/20
30/30 [==============================] - 1s 35ms/step - loss: 0.0920 - accuracy: 0.9757 - val_loss: 0.3341 - val_accuracy: 0.8733
Epoch 9/20
30/30 [==============================] - 1s 38ms/step - loss: 0.0745 - accuracy: 0.9826 - val_loss: 0.3258 - val_accuracy: 0.8815
Epoch 10/20
30/30 [==============================] - 1s 38ms/step - loss: 0.0649 - accuracy: 0.9838 - val_loss: 0.3468 - val_accuracy: 0.8812
Epoch 11/20
30/30 [==============================] - 1s 38ms/step - loss: 0.0515 - accuracy: 0.9901 - val_loss: 0.3663 - val_accuracy: 0.8797
Epoch 12/20
30/30 [==============================] - 1s 38ms/step - loss: 0.0416 - accuracy: 0.9923 - val_loss: 0.3964 - val_accuracy: 0.8778
Epoch 13/20
30/30 [==============================] - 1s 35ms/step - loss: 0.0354 - accuracy: 0.9941 - val_loss: 0.4131 - val_accuracy: 0.8740
Epoch 14/20
30/30 [==============================] - 1s 38ms/step - loss: 0.0277 - accuracy: 0.9953 - val_loss: 0.4368 - val_accuracy: 0.8747
Epoch 15/20
30/30 [==============================] - 2s 59ms/step - loss: 0.0237 - accuracy: 0.9961 - val_loss: 0.4633 - val_accuracy: 0.8743
Epoch 16/20
30/30 [==============================] - 2s 73ms/step - loss: 0.0163 - accuracy: 0.9983 - val_loss: 0.5578 - val_accuracy: 0.8661
Epoch 17/20
30/30 [==============================] - 1s 39ms/step - loss: 0.0126 - accuracy: 0.9991 - val_loss: 0.5390 - val_accuracy: 0.8703
Epoch 18/20
30/30 [==============================] - 1s 38ms/step - loss: 0.0107 - accuracy: 0.9990 - val_loss: 0.5594 - val_accuracy: 0.8705
Epoch 19/20
30/30 [==============================] - 1s 34ms/step - loss: 0.0082 - accuracy: 0.9994 - val_loss: 0.5882 - val_accuracy: 0.8707
Epoch 20/20
30/30 [==============================] - 1s 37ms/step - loss: 0.0060 - accuracy: 0.9995 - val_loss: 0.6376 - val_accuracy: 0.8600
```

Now let's analyse the loss and metrics through tensorboard

```
%load_ext tensorboard
%tensorboard --logdir /tensorboard_files
```




```
# Evaluate model that is trained for 20 epochs
model.evaluate(x_test, y_test)
```

```
782/782 [==============================] - 2s 3ms/step - loss: 0.7122 - accuracy: 0.8442
```

```
[0.7121971249580383, 0.8442400097846985]
```

The above model is not the best model. It is overfitted! How do we know? (More on this in the Sec.3 of notebook)

Validation loss is not minimum and has increased alot at 20th epoch.

So, how do we get back to a model at the epoch with the minimum validation loss?

Simple. **Load the model saved by ModelCheckpointing!**

```
# Load saved best model based
best_model = tf.keras.models.load_model("imdb_model_checkpoint")
best_model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 16)                160016

 dense_1 (Dense)             (None, 16)                272

 dense_2 (Dense)             (None, 1)                 17

=================================================================
Total params: 160,305
Trainable params: 160,305
Non-trainable params: 0
_________________________________________________________________
```

Let's evaluate the saved model on the test data.

```
# Evaluate model
best_model.evaluate(x_test, y_test)
```

```
782/782 [==============================] - 2s 3ms/step - loss: 0.2947 - accuracy: 0.8845
[0.2947026491165161, 0.8845199942588806]
```

Q: Compare the test accuracy of best_model with model.

2. K-fold Validation

What if we have a small data set for training (say 500 samples) ?

It would mean our validation set would be even smaller (say 100 samples)!

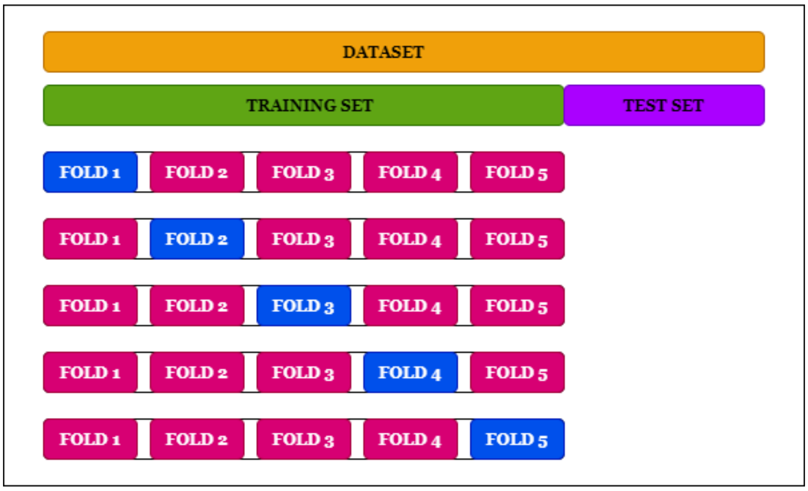
As a consequence, the validation scores will **vary** depending upon which 100 samples out of the original training data are included in the validation set.

Consequently, the validation scores will have a high variance with regard to the validation split.

In such a scenario, HOW do we reliably evaluate the model?

Solution - **K-fold cross-validation**

Here's an exmaple with $k = 5$



For convenience, let's define a function called build_model(), wherein we define a model and compile it. We are doing this because we will need to call this function multiple times later on.

```
# Function to build a sequential model
def build_model():
    model = keras.Sequential([
            layers.Dense(32, activation='relu'),
            layers.Dense(32, activation='relu'),
            layers.Dense(1, activation='sigmoid')
    ])
    model.compile(loss="binary_crossentropy", optimizer="rmsprop", metrics=["accuracy"])
    return model
```

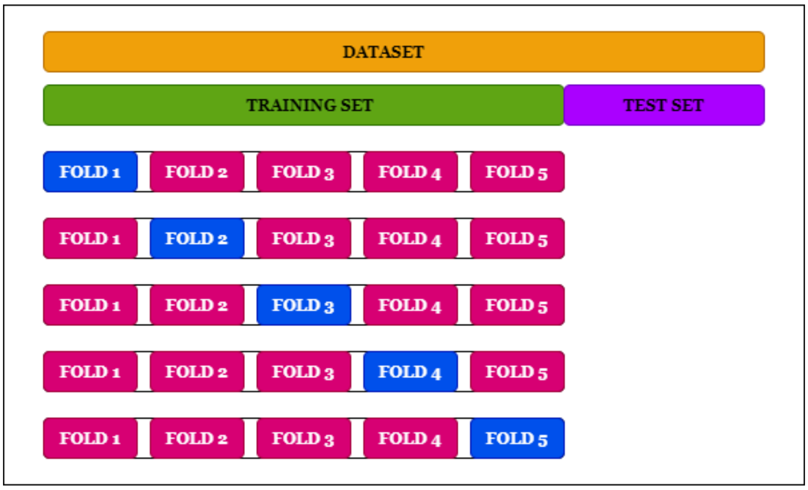
```
# Load data
(train_data, train_labels), (test_data, test_labels) = imdb.load_data(num_words=10000)
x_train = vectorize_sequences(train_data)
x_test = vectorize_sequences(test_data)
y_train = np.asarray(train_labels).astype("float32")
y_test = np.asarray(test_labels).astype("float32")
```

Now for the sake of demonstration

We are reducing the number of available training samples.

In other words, **imagine that you just had 1000 samples in your training set**

```
# Using reduced no. of samples from the training data to simulate a situation
# where less data is available - hence, the need of k-fold cross validation

n_reduced_samples = 1000
x_train = x_train[0:n_reduced_samples,:]
y_train = y_train[0:n_reduced_samples]
# print(y_train)
```




```
# K-fold validation due to less data
k = 4       # 4- fold validation
num_val_samples = len(x_train)//k  #Q: What is len(x_train)
num_epochs = 30

# lists to store hisories of each fold
# note: if "_train_" not mentioned, then it is a validation metric
all_accuracy_histories = []
all_loss_histories = []
all_train_loss_histories = []
all_train_accuracy_histories = []

# looping over each split
for i in range(k):
    print(f"processing split {i}")
    # Split out validation set
    x_val = x_train[i * num_val_samples: (i + 1) * num_val_samples]
    y_val = y_train[i * num_val_samples: (i + 1) * num_val_samples]

    # Concatenate the training samples on the left and right side of the val samples
```

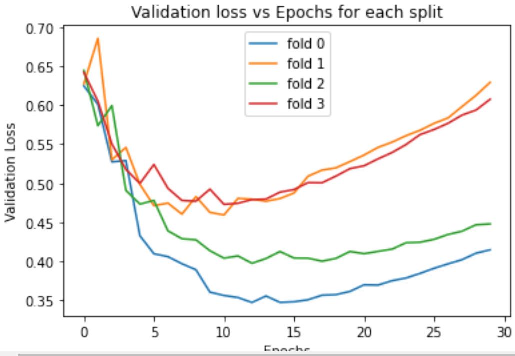
```
    partial_x_train = np.concatenate(
                        [x_train[:i * num_val_samples],
                        x_train[(i + 1) * num_val_samples:]],
                        axis=0)
    partial_y_train = np.concatenate(
                        [y_train[:i * num_val_samples],
                        y_train[(i + 1) * num_val_samples:]],
                        axis=0)
    # Q: Why axis=0?

    # build the model and fit with fold-specific data
    model = build_model()
    history = model.fit(partial_x_train, partial_y_train, validation_data=(x_val, y_val),epochs=num_epochs, batch_size=256, verbose=0)

    # manually extracting metrics to perform custom visualization later
    # Q: What is history.history ?
    accuracy_history = history.history["val_accuracy"]
    loss_history = history.history["val_loss"]
    train_loss_history = history.history["loss"]
    train_accuracy_history = history.history["accuracy"]

    # Append history data to the fold-level list
    all_accuracy_histories.append(accuracy_history)
    all_loss_histories.append(loss_history)
    all_train_loss_histories.append(train_loss_history)
    all_train_accuracy_histories.append(train_accuracy_history)

# Q: what would be the len(all_xx_histories) and len(xx_history)
```

```
processing split 0
processing split 1
processing split 2
processing split 3
```

Now, let's visualize the losses and accuracies for each fold.

```
# Plot validation losses vs epoch for each split
for i,entry in enumerate(all_loss_histories):
  label="fold " + str(i)
  plt.plot(entry, label=label)

plt.xlabel("Epochs")
plt.ylabel("Validation Loss")
plt.title("Validation loss vs Epochs for each split")
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7f1e57ac7ee0>
```



The curves are significantly different! This shows us that -

for small datasets, the validation loss can have HIGH variance with regard to the validation split.

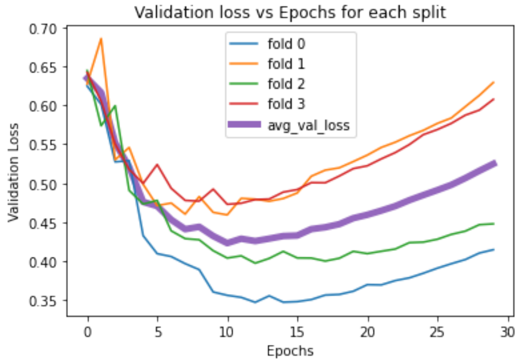
A more robust measure of the validation loss is to use the avarege of the validation loss across the 4 folds.

Let's compute and plot the avg validation loss

```
# Plot validation losses vs epoch for each fold
for i,entry in enumerate(all_loss_histories):
  label="fold " + str(i)
  plt.plot(entry, label=label)

# compute and plot the avg val loss across folds
loss_histories_matrix = np.array(all_loss_histories)
print(f"histories_matrix.shape = {loss_histories_matrix.shape}")
avg_loss = loss_histories_matrix.mean(axis=0)
```

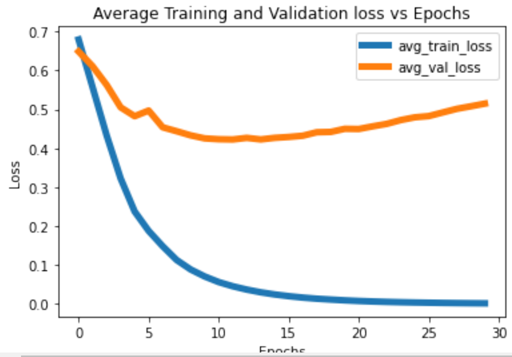
```
plt.plot(avg_loss, label='avg_val_loss', linewidth=5, zorder=-10)
plt.xlabel("Epochs")
plt.ylabel("Validation Loss")
plt.title("Validation loss vs Epochs for each split")
plt.legend()

# compute the epoch at which the teh avg validation loss is min
best_epoch = np.argmin(avg_loss)
print(f"minimum val loss at epoch: {best_epoch}")
```

```
histories_matrix.shape = (4, 30)
minimum val loss at epoch: 10
```



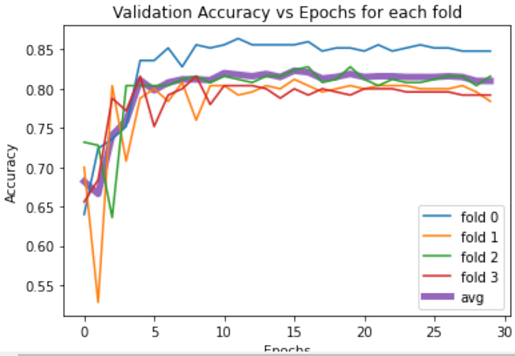

```
# Compute avg training loss and plot it
train_loss_histories_matrix = np.array(all_train_loss_histories)
avg_train_loss = train_loss_histories_matrix.mean(axis=0)
plt.plot(avg_train_loss, label='avg_train_loss', linewidth=5, zorder=-10)
plt.plot(avg_loss, label='avg_val_loss',linewidth=5, zorder=-10)
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.title("Average Training and Validation loss vs Epochs")
plt.legend()

# Compute avg training loss
train_accuracy_histories_matrix = np.array(all_train_accuracy_histories)
avg_train_accuracy = train_accuracy_histories_matrix.mean(axis=0)
print(f"avg_train_loss at {best_epoch} (best) epoch = {avg_train_loss[best_epoch]}")
print(f"avg_train_accuracy at {best_epoch} (best) epoch = {avg_train_accuracy[best_epoch]}")
```

```
avg_train_loss at 11 (best) epoch = 0.04534864518791437
avg_train_accuracy at 11 (best) epoch = 1.0
```




```
# Plot validation accuracy for each fold
for i,entry in enumerate(all_accuracy_histories):
  label="fold " + str(i)
  plt.plot(entry, label=label)

# Compute and plot avg validation accuracy
histories_matrix = np.array(all_accuracy_histories)
print(f"histories_matrix.shape = {histories_matrix.shape}")
avg_accuracy = histories_matrix.mean(axis=0)
plt.plot(avg_accuracy, label='avg', linewidth=5, zorder=-10)

plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.title("Validation Accuracy vs Epochs for each fold")
plt.legend()
```

```
histories_matrix.shape = (4, 30)
<matplotlib.legend.Legend at 0x7f1e57c71cd0>
```



Now that we know we know at what epoch we start overfitting, we will retrain our model to the best_epoch number of epochs and get a robust fit.

For the sake of being explicit, we are building a new model called final_model.

```
# Train on the whole training with best no. of epochs to get the final model
final_model = build_model()
history = final_model.fit(x_train, y_train, epochs=best_epoch+1, batch_size=256, verbose=1)
# Q: What loss and accuracy are you seeing below? - Training/Validation/Testing?
```

```
Epoch 1/11
4/4 [==============================] - 1s 18ms/step - loss: 0.6845 - accuracy: 0.5510
Epoch 2/11
4/4 [==============================] - 0s 18ms/step - loss: 0.5022 - accuracy: 0.9060
Epoch 3/11
4/4 [==============================] - 0s 18ms/step - loss: 0.4031 - accuracy: 0.8830
Epoch 4/11
4/4 [==============================] - 0s 18ms/step - loss: 0.2840 - accuracy: 0.9840
Epoch 5/11
4/4 [==============================] - 0s 18ms/step - loss: 0.2283 - accuracy: 0.9850
Epoch 6/11
4/4 [==============================] - 0s 20ms/step - loss: 0.1709 - accuracy: 0.9960
Epoch 7/11
4/4 [==============================] - 0s 18ms/step - loss: 0.1293 - accuracy: 0.9980
Epoch 8/11
4/4 [==============================] - 0s 18ms/step - loss: 0.1011 - accuracy: 0.9990
Epoch 9/11
4/4 [==============================] - 0s 18ms/step - loss: 0.0777 - accuracy: 0.9980
Epoch 10/11
4/4 [==============================] - 0s 20ms/step - loss: 0.0607 - accuracy: 1.0000
Epoch 11/11
4/4 [==============================] - 0s 18ms/step - loss: 0.0508 - accuracy: 0.9990
```

Q: Compare the xx loss and accuracies with the avg xx loss and accuracy during k-fold-validation

```
# Evaluate final model on test set
final_model.evaluate(x_test, y_test)
```

```
782/782 [==============================] - 2s 3ms/step - loss: 0.3961 - accuracy: 0.8318
[0.39610257744789124, 0.8317599892616272]
```

There is one more validation method called Iteratad K-fold cross validation.

For P iterations:

- Reshuffle data
- Perform K-fold cross validation Take avg of all scores.

Q: How many models do you end up training?

Q: $avg = \frac{sum}{n}$. What is n ?

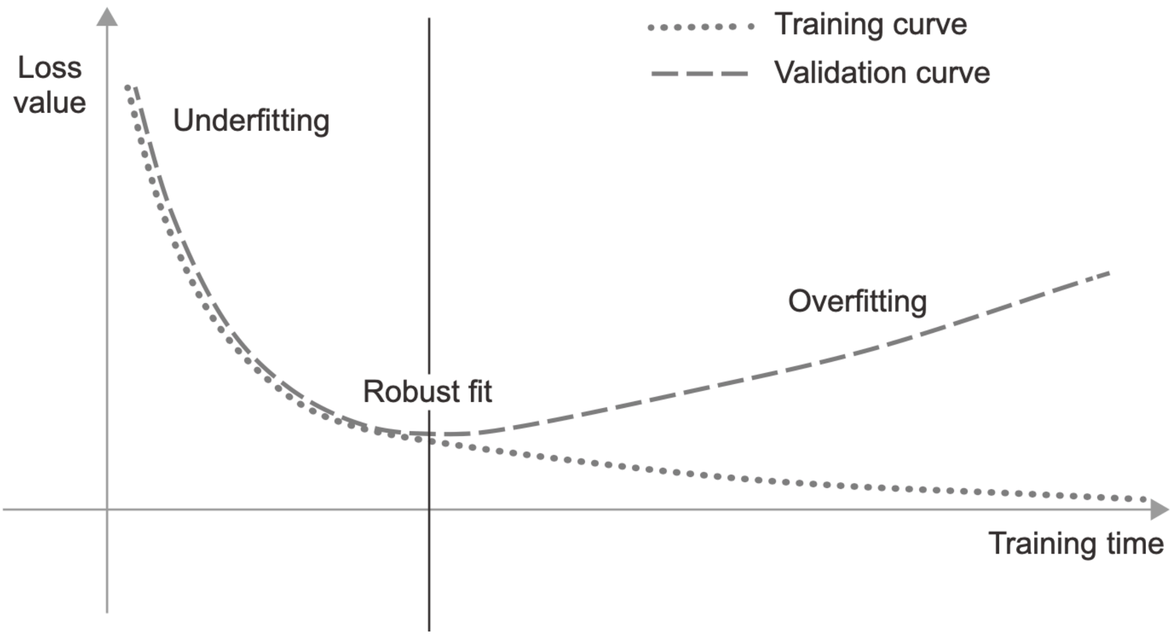
3. Regularization

Regularization techniques are a set of best practices that actively impede the model's ability to fit perfectly to the training data, with the goal of making the model perform better during validation.

This is called "regularizing" the model, because it tends to make the model simpler, more "regular," its curve smoother, more "generic";

thus it is less specific to the training set and better able to generalize by more closely approximating the latent manifold of the data.

The goal of machine learning is to get a model that generalise well. Such models provide a robust fit.




```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras import regularizers


# define the model
def get_neural_network(train_data,layers,regularization,dropout):
  """
  Builds and trains a neural network based on arguments and predefined template
  Returns: history
  train_data: Tuple (x_train, y_train)
  layers: dictionary of layers; e.g. {"h1":[16, "relu"],"h2":[16,"relu"],"output":[1,"sigmoid"]}
  regularization: True/False
  dropout: True/False
  """

  model = Sequential()
  # Add layers based on conditions
  for i,j in enumerate(layers.keys()):   # Q: layers.keys() returns ?
    if regularization and i != len(layers)-1:    # reg == True and not the last (output) layer
      model.add(Dense(layers[j][0], activation=layers[j][1],    # j is a key. e.g. "h1"
                kernel_regularizer=regularizers.l2(0.002)))    # add 0.002 * w**2 for each w in W to the loss

    else:
      model.add(Dense(layers[j][0], activation=layers[j][1]))

    if dropout and i == 0:   # if dropout == True and it is the first layer
      model.add(Dropout(0.5))           # add Dropout layer

  # Compile the model
  model.compile(optimizer="rmsprop",
                loss="binary_crossentropy",
                metrics=["accuracy"])
  # Fit the model
  x_train, y_train = train_data
  history = model.fit(x_train,y_train,
                      epochs=20,
                      validation_split=0.4,
                      batch_size=512,
                      verbose=1)
  return history
```

Note: During regularization, because the penalty is only added at training time, the loss for this model will be much higher at training than at test time

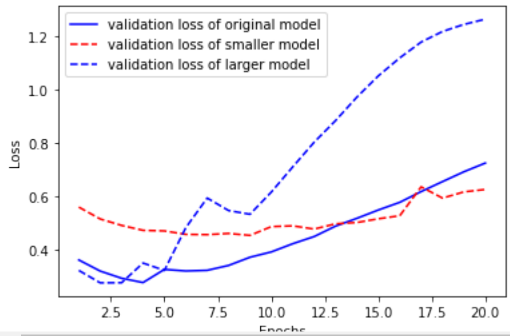
```
# Load data
(train_data, train_labels), (test_data, test_labels) = imdb.load_data(num_words=10000)
```

```
x_train = vectorize_sequences(train_data)
x_test = vectorize_sequences(test_data)
y_train = np.asarray(train_labels).astype("float32")
y_test = np.asarray(test_labels).astype("float32")

train_data = (x_train, y_train)


# layers for different models
original_layers = {"h1":[16, "relu"],"h2":[16,"relu"],"output":[1,"sigmoid"]}
smaller_layers = {"h1":[4, "relu"],"h2":[4,"relu"],"output":[1,"sigmoid"]}
larger_layers = {"h1":[512, "relu"],"h2":[512,"relu"],"output":[1,"sigmoid"]}


# get different trained networks for comparison
history_smaller_model = get_neural_network(train_data,smaller_layers,False,False) # model with lower capacity
history_larger_model = get_neural_network(train_data,larger_layers,False,False) # model with higher capacity
history_original_model = get_neural_network(train_data,original_layers,False,False) # original model

history_regularization = get_neural_network(train_data,original_layers,True,False) # model with L2 regularization
history_dropout = get_neural_network(train_data,original_layers,False,True) # model with Dropout layers
```

```
Epoch 1/20
30/30 [==============================] - 2s 49ms/step - loss: 0.6417 - accuracy: 0.6495 - val_loss: 0.5961 - val_accuracy: 0.6628
Epoch 2/20
30/30 [==============================] - 1s 35ms/step - loss: 0.5574 - accuracy: 0.7868 - val_loss: 0.5377 - val_accuracy: 0.8266
Epoch 3/20
30/30 [==============================] - 1s 33ms/step - loss: 0.5046 - accuracy: 0.8398 - val_loss: 0.5114 - val_accuracy: 0.7857
Epoch 4/20
30/30 [==============================] - 1s 32ms/step - loss: 0.4684 - accuracy: 0.8721 - val_loss: 0.4847 - val_accuracy: 0.8418
Epoch 5/20
30/30 [==============================] - 1s 32ms/step - loss: 0.4404 - accuracy: 0.8942 - val_loss: 0.4683 - val_accuracy: 0.8703
Epoch 6/20
30/30 [==============================] - 1s 31ms/step - loss: 0.4183 - accuracy: 0.9124 - val_loss: 0.4589 - val_accuracy: 0.8610
Epoch 7/20
30/30 [==============================] - 1s 32ms/step - loss: 0.3991 - accuracy: 0.9240 - val_loss: 0.4545 - val_accuracy: 0.8546
Epoch 8/20
30/30 [==============================] - 1s 31ms/step - loss: 0.3821 - accuracy: 0.9357 - val_loss: 0.4416 - val_accuracy: 0.8761
Epoch 9/20
30/30 [==============================] - 1s 32ms/step - loss: 0.3669 - accuracy: 0.9465 - val_loss: 0.4440 - val_accuracy: 0.8659
Epoch 10/20
30/30 [==============================] - 1s 31ms/step - loss: 0.3534 - accuracy: 0.9531 - val_loss: 0.4451 - val_accuracy: 0.8618
Epoch 11/20
30/30 [==============================] - 1s 31ms/step - loss: 0.3406 - accuracy: 0.9594 - val_loss: 0.4441 - val_accuracy: 0.8633
Epoch 12/20
30/30 [==============================] - 1s 35ms/step - loss: 0.3290 - accuracy: 0.9637 - val_loss: 0.4393 - val_accuracy: 0.8666
Epoch 13/20
30/30 [==============================] - 1s 32ms/step - loss: 0.3181 - accuracy: 0.9669 - val_loss: 0.4400 - val_accuracy: 0.8677
Epoch 14/20
30/30 [==============================] - 1s 32ms/step - loss: 0.3079 - accuracy: 0.9704 - val_loss: 0.4465 - val_accuracy: 0.8634
Epoch 15/20
30/30 [==============================] - 1s 33ms/step - loss: 0.2985 - accuracy: 0.9731 - val_loss: 0.4432 - val_accuracy: 0.8678
Epoch 16/20
30/30 [==============================] - 1s 32ms/step - loss: 0.2894 - accuracy: 0.9751 - val_loss: 0.4838 - val_accuracy: 0.8460
Epoch 17/20
30/30 [==============================] - 1s 32ms/step - loss: 0.2802 - accuracy: 0.9784 - val_loss: 0.4312 - val_accuracy: 0.8759
Epoch 18/20
30/30 [==============================] - 1s 33ms/step - loss: 0.2718 - accuracy: 0.9805 - val_loss: 0.4319 - val_accuracy: 0.8748
Epoch 19/20
30/30 [==============================] - 1s 35ms/step - loss: 0.2640 - accuracy: 0.9813 - val_loss: 0.4588 - val_accuracy: 0.8650
Epoch 20/20
30/30 [==============================] - 1s 32ms/step - loss: 0.2565 - accuracy: 0.9825 - val_loss: 0.4735 - val_accuracy: 0.8593
Epoch 1/20
30/30 [==============================] - 10s 310ms/step - loss: 0.5511 - accuracy: 0.7476 - val_loss: 0.3030 - val_accuracy: 0.88
Epoch 2/20
30/30 [==============================] - 9s 297ms/step - loss: 0.2657 - accuracy: 0.8987 - val_loss: 0.2744 - val_accuracy: 0.892
Epoch 3/20
30/30 [==============================] - 9s 293ms/step - loss: 0.1471 - accuracy: 0.9455 - val_loss: 0.3114 - val_accuracy: 0.886
Epoch 4/20
30/30 [==============================] - 9s 296ms/step - loss: 0.1214 - accuracy: 0.9663 - val_loss: 0.3844 - val_accuracy: 0.875
Epoch 5/20
30/30 [==============================] - 10s 320ms/step - loss: 0.0798 - accuracy: 0.9805 - val_loss: 0.3002 - val_accuracy: 0.88
Epoch 6/20
30/30 [==============================] - 10s 341ms/step - loss: 0.0079 - accuracy: 0.9995 - val_loss: 0.4916 - val_accuracy: 0.88
Epoch 7/20
30/30 [==============================] - 9s 298ms/step - loss: 0.0012 - accuracy: 0.9998 - val_loss: 0.6504 - val_accuracy: 0.881
Epoch 8/20
30/30 [==============================] - 10s 321ms/step - loss: 0.2954 - accuracy: 0.9734 - val_loss: 0.5201 - val_accuracy: 0.88
Epoch 9/20
```

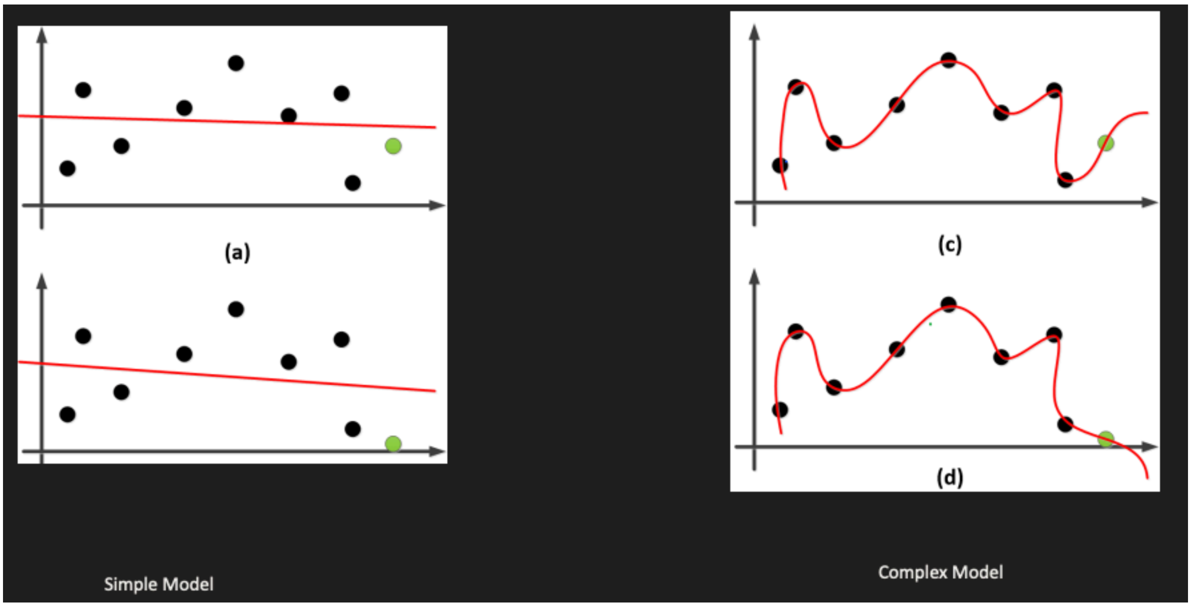
```
val_loss_original = history_original_model.history["val_loss"]
val_loss_smaller = history_smaller_model.history["val_loss"]
val_loss_larger = history_larger_model.history["val_loss"]
epochs = range(1,21)
plt.plot(epochs,val_loss_original,"b",label="validation loss of original model")
plt.plot(epochs,val_loss_smaller,"r--",label="validation loss of smaller model")
```

```
plt.plot(epochs,val_loss_larger,"b--",label="validation loss of larger model")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()
plt.show()
```



1. The smaller model starts overfitting later than the reference model (after 10 epochs rather than 4), and its performance degrades more slowly once it starts overfitting.
2. The bigger model starts overfitting almost immediately, after just one/two epoch, and it overfits much more severely. Its validation loss is also noisier. It gets training loss near zero very quickly. The more capacity the model has, the more quickly it can model the training data (resulting in a low training loss), but the more susceptible it is to overfitting (resulting in a large difference between the training and validation loss).

WHY?

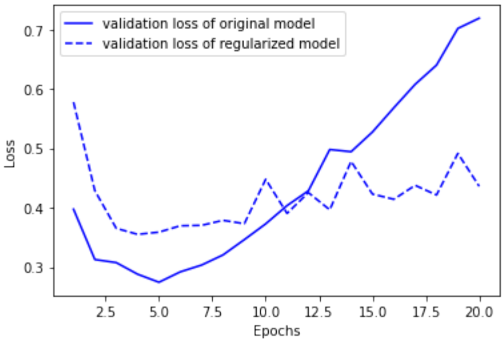


We have seen the effects of model capacity (or size). Now let's see the effect of some regularization techniques.

First, we will see **weight regularization**

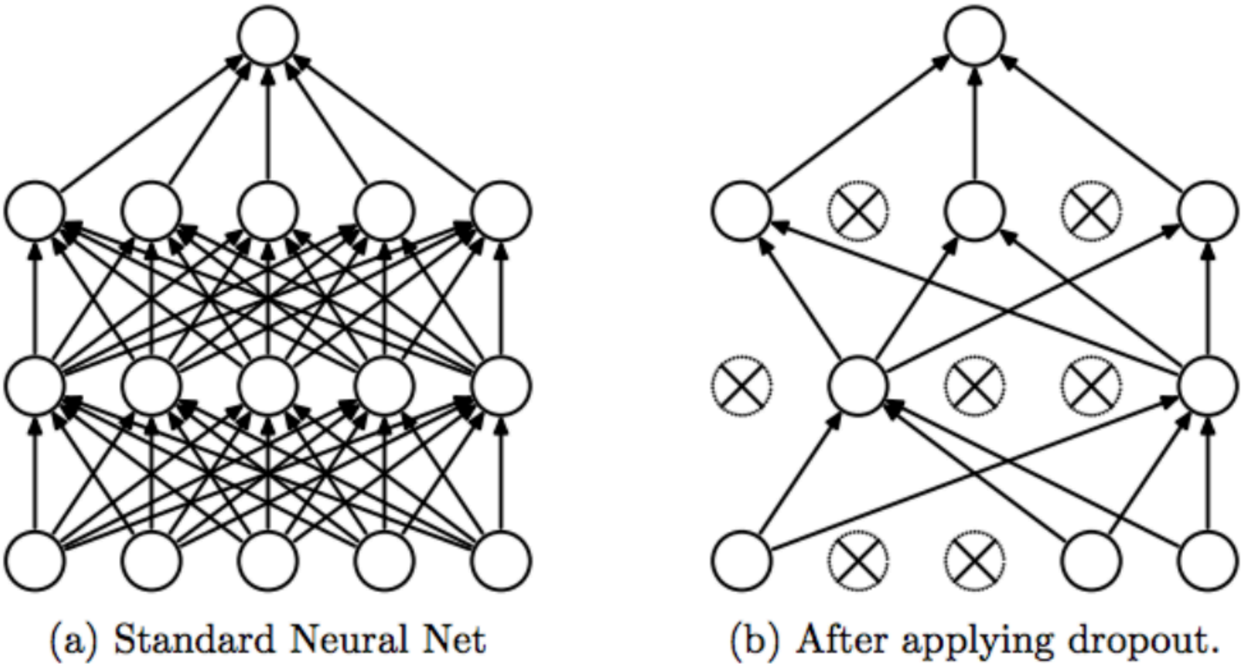
Remember, we have used L2(0.002) regularization. (Q: what is 0.002?)

```
val_loss_original = history_original_model.history["val_loss"]
val_loss_regularization = history_regularization.history["val_loss"]
epochs = range(1,21)
plt.plot(epochs,val_loss_original,"b",label="validation loss of original model")
plt.plot(epochs,val_loss_regularization,"b--",label="validation loss of regularized model")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()
plt.show()
```

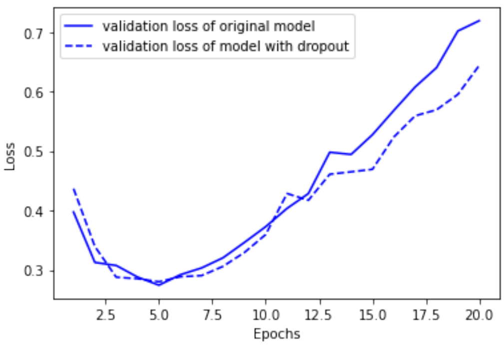


The model with L2 regularization has become much more resistant to overfitting than the reference model, even though both models have the same number of parameters.

Now let's consider another regularziation technique - **Adding Dropout**




```
val_loss_original = history_original_model.history["val_loss"]
val_loss_dropout = history_dropout.history["val_loss"]
epochs = range(1,21)
plt.plot(epochs,val_loss_original,"b",label="validation loss of original model")
plt.plot(epochs,val_loss_dropout,"b--",label="validation loss of model with dropout")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()
plt.show()
```



We see that the validation loss with dropout is lower than that of the original model.

Dropout will have a more significant effect for larger models.

Start coding or generate with AI.

Start coding or generate with AI.