

Training Workflows for Deep Learning

Department of Electrical Engineering
University of Notre Dame, USA

week 6,7,8 (updated: November 8, 2023)

Deep Learning Workflows

- We want to train a neural network model that can *generalize* beyond its training data.
- Deep neural networks, however, have a very large VC dimension and this suggests that such models can *memorize* all of the samples in the data set and yet perform poorly on samples that are not in the training data. We say such models *overfit* the training data.
- On the other hand, selecting a model that performs poorly on the training data is said to *underfit* the training data.
- The main problem in model training is to find that sweet spot between underfitting and overfitting.

Deep Learning Workflows

- The training process is a series of *experiments* whose results guide one in searching through the model set for this sweet spot.
- This series of experiments form the *training workflow*.

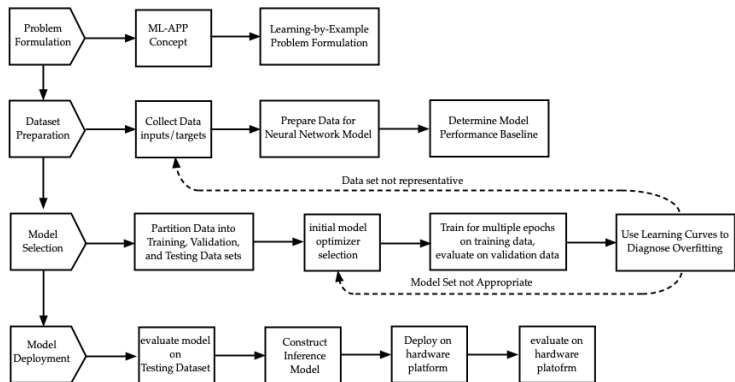


Figure: Deep Learning Application Development Workflow

Deep Learning Workflows

- **Problem Formulation** takes a customer's app concept and maps it to the learning-by-example problem described in chapter ??.
- **Data Preparation** involves collecting and preparing the data used in the neural network model. We use the data to identify a “baseline” performance level that our trained model will need to beat. This stage of the training process also partitions the available data in a partial or *p-training* dataset, *validation* dataset and *test* dataset.
- **Model Selection** starts with an assumed model architecture and optimizer configuration and then uses a mini-batch optimizer on the *p-training* dataset to fit (train) the model for several epochs. The model's loss and performance metric are evaluated after each training epoch on the *p-training* and validation datasets. The resulting training history is plotted as a function of epoch to form that training session's *learning curves*.

Deep Learning Workflows

- **Model Modification:** The learning curves are used to diagnose whether the model is overfitting the p-training data and that diagnosis is used to suggest changes to the model architecture, p-training and validation datasets, optimizer hyper-parameters, and the problem's reward function. We then re-train the modified model to generate a new set of learning curves and proceed in this matter until we have a model that no longer overfits the p-training data and whose performance metric on the validation dataset is deemed to be acceptable.
- **Model Deployment** is the final stage of our workflow. This stage evaluates the model's performance on the test dataset to assess how well the model should work in practice. We then prepare an *inference* version of our model that strips out the training portion of the code and rewrites that model to run on the platform used in our application. This inference version of the model is then evaluated in the field to see how well it actually performs with respect to the training model's performance on the test dataset.

Problem Formulation

Problem Formulation forces the developer to clearly identify how the customer's application maps to the learning-by-example framework. We illustrate using the handwritten digit recognition application.

- The customer is the USPS and they want an app scans the handwritten address on a letter and uses that scanned image to automatically sort where the letter should go.
- The basic tasks of this problem are:
 - ① segmenting out each character in the address
 - ② Classifying which the character
 - ③ outputting a string for the address.
- We focus on the problem of classifying a single scanned digit, assuming it is a digit $0, 1, \dots, 9$.
- The customer asks for a prototype application of this streamlined problem with classification accuracy of at least 99%.

Problem Formulation

Not all problems are deep learning problems. We need to map our problem statement to *learning by example*

- **System:** the system generates a data set $\mathcal{D} = \{x_k, y_k\}_{k=1}^N$ of N samples. The input sample, x_k , is the scanned image of a digit. The target sample is the classification of the input image x_k as a digit $(0, 1, \dots, 9)$ where the classification was performed by a human *observer*.
- The *model set*, \mathcal{H} , consists of sequential neural network models, $h : \mathcal{X} \rightarrow [0, 1]^{10}$, that map scan images, $x \in \mathcal{X}$, of handwritten digits onto a real 10-dimensional vector, $h(x)$, whose components are real numbers in the interval $[0, 1]$.
- The *loss function* will be *sparse categorical cross entropy function*

$$L(y_k, h(x_k)) = -\frac{1}{10} \sum_{j=1}^{10} \mathbb{1}(y_k = j - 1) \log(h_j(x_k))$$

- Our *performance metric* is classification accuracy.

Data Preparation

- Data preparation is one of the most time consuming stages in developing a DL application. This stage involves the collection of the data as well as pre-processing the data so it can be directly used by a neural network model.
- This stage also includes partitioning the data into a p-training, validation, and testing dataset, as well as generating a Dataset object to pre-batch the datasets.
- This stage often includes generating a *baseline* performance metric from the training data, that represents the metric to beat.
- We show how these steps for the MNIST database problem.

MNIST database

- MNIST is a large database of scanned images of handwritten digits (0 – 9). It has 60,000 training images and 10,000 testing images.
- All input samples are 28×28 monochrome images of a digit whose pixels are uint8.
- The target is an integer between 0, ..., 9.
- The database can be obtained from TensorFlow

PYTHON SCRIPT

```
from tensorflow.keras.datasets import mnist
(train_x, train_y), (test_x, test_y) = mnist.load_data()

import numpy as np
print(f"train_x shape = {train_x.shape}")
print(f"train_y shape = {train_y.shape}")
rindx = np.ceil(np.random.uniform(60000)-1).astype("int64")
print(f"sample {rindx} out of 60000")

input = train_x[rindx,:,:]
target = train_y[rindx]

import matplotlib.pyplot as plt
plt.imshow(input, cmap='Greys')
plt.axis("off")
plt.title(f"Target Label = {target}, Sample = {rindx}")
```

OUTPUT

```
train_x shape = (60000, 28, 28)
train_y shape = (60000,)
sample 20355 out of 60000
```



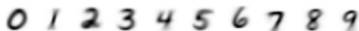
- Note that the input tensor's shape is (60000, 28, 28), so our input is a 28 by 28 pixel image with only a single channel. The data type is uint8.
- We will be using a dense model that expects a floating point input which is a vector. So we need to reshape the input image into a vector.

```
print(f"train_x shape = {train_x.shape}, train_x dtype = {train_x.dtype}")
train_x = train_x.reshape(60000, 28*28).astype("float32")/255
test_x = test_x.reshape(10000, 28*28).astype("float32")/255
print(f"train_x shape = {train_x.shape}, train_x dtype = {train_x.dtype}")
```

- *Baseline model* whose performance is easily evaluated on the training data.
- The baseline model's performance represents what our actual model must beat if we are to demonstrate that the model learned anything during training.
- For MNIST, the baseline is the "average" image computed for each class.

```
import matplotlib.pyplot as plt
class_cnt = np.zeros(10)
class_freq = np.zeros(10)
baseline_maps = np.zeros(shape=(10,28,28))
for indx in range(db_size):
    target = train_y[indx]
    class_cnt[target]+=1
for indx in range(db_size):
    input = train_x[indx,:,:]
    target = train_y[indx]
    baseline_maps[target,:,:] += input/class_cnt[target]

figure, axis = plt.subplots(1,10)
for k in range(10):
    baseline = baseline_maps[k,:,:]
    axis[k].imshow(baseline, cmap="Greys")
    axis[k].set_axis_off()
```



0 1 2 3 4 5 6 7 8 9

Data Preparation

- Classify each image in dataset by seeing which baseline it is closest to.
- Distance is measured by RMSE or MAE

$$\text{RMSE}(k) = \sqrt{\frac{1}{N} \sum_{i=1}^N (\hat{x}_k - x_k)^2}$$

$$\text{MAE}(k) = \frac{1}{N} \sum_{i=1}^N |\hat{x}_k - x_k|$$

- Best average performance of baseline is 47% with respect to MAE.

norm	total_acc	0	1	2	3	4	5	6	7	8	9
RMSE	81%	87%	97%	76%	77%	82%	64%	86%	83%	71%	79%
MAE	65%	82%	99%	45%	60%	68%	25%	77%	76%	38%	73%

Table: MNIST Baseline Model's Accuracy Performance

Data Preparation

- Final step is to construct a Dataset object of the training and testing data.
- Dataset objects are iterators that pre-batch the data so it is easier for our training method (`fit`) to fetch a batch of data used by backpropagation.
- We fetch a single batch from the Dataset object and check its shape. Input batches are (32, 784) and target is (32,).

```
import tensorflow as tf
train_ds = tf.data.Dataset.from_tensor_slices((train_x, train_y))
train_ds = train_ds.batch(32)
test_ds = tf.data.Dataset.from_tensor_slices((test_x, test_y))
test_ds = test_ds.batch(32)

for input,target in train_ds:
    print(f"shape of batched input = {input.shape}")
    print(f"shape of batched target = {target.shape}")
    break
```

Model Selection

- Model selection is concerned with training a model that has good performance metric while not overfitting the training data.
- This is an *experimental* process in which each training session is viewed as an experiment. The experiment's hypothesis is that a specified model architecture and training configuration will have good performance and little overfitting.
- If the results invalidate the hypothesis, then we modify the model architecture or training configuration to achieve a better outcome in the next training session.
- We continue in this manner until we are satisfied that the model performs well and does not overfit.

Model Selection

- The first step is to partition the available training data into a partial or *p-training* data set and a *validation* data set.
- We then instantiate the model as a TensorFlow `Model` and use the `compile` method to configure the training optimizer.
- Finally we use the `fit` method to train the model for a fixed number of *training epochs*.
- At each epoch, the `fit` model computes the loss and performance metrics for the *p-training* and validation datasets.
- We then plot the *p-training/validation* curves to assess whether model performance is acceptable and whether it is overfitting the training data.

Model Selection

- If we do not find an acceptable model in a training session, then we either change the training data, model architecture, or reconfigure the optimizer, and then try again.
- Model architecture may be modified by changing number of layers, type of layers, number of nodes in each layer, changing activation functions
- Changing the data is usually done by increasing the amount of training data. This can be accomplished by changing the validation split or using a more sophisticated k -fold cross validation scheme.
- Reconfiguring the optimizer involves adjusting the type of optimizer (RMSprop versus Adam), the optimizer's hyper-parameters (learning-rate, momentum weight, etc).
- One can also change the loss function being used by adding in some sort of regularization
- There are many ways this adjustment can be made and this means the model selection process is very time consuming.

Model Selection (MNIST)

- Model selection for the MNIST application starts by instantiating a sequential model and then training that model for several epochs.
- In this case, I started with a subset of MNIST data (10000 training and testing samples)
- I then construct Dataset objects for p-training and validations assuming a batch size of 32.
- I assumed a 10% validation split.

```
from tensorflow.keras.datasets import mnist
import tensorflow as tf

(train_x, train_y), (test_x, test_y) = mnist.load_data()
train_x = train_x[:10000, :, :]
train_x = train_x.reshape(10000, 28*28).astype("float32")/255
train_y = train_y[:10000]

train_ds = tf.data.Dataset.from_tensor_slices((train_x, train_y))
train_ds.batch(32)

validation_split = 0.10
train_ds_size = len(list(train_ds))
val_ds_size = int(val_split*train_ds_size)
ptrain_ds_size = train_ds_size-val_ds_size

ptrain_ds = train_ds.take(ptrain_ds_size)
val_ds = train_ds.skip(ptrain_ds_size).take(val_ds_size)
```

Model Selection - MNIST

- Select sequential model with two Dense layers of 512 and 64 nodes, respectively, using ReLu activation.
- Output layer is Dense with 10 nodes and softmax activation.
- Input layer has $28 \times 28 = 784$ nodes with linear activation.
- We compile model using RMSprop optimizer (default settings) and declare a sparse categorical cross-entropy loss function.

```
from tensorflow import keras
from tensorflow.keras import layers

inputs = keras.Input(shape=(28*28))
x      = layers.Dense(512, activation="relu")(inputs)
x      = layers.Dense(64, activation="relu")(x)
outputs = layers.Dense(10, activation="softmax")(x)
model = keras.Model(inputs=inputs, outputs=outputs)

model.compile(optimizer="rmsprop",
              loss = "sparse_categorical_crossentropy",
              metrics = ["accuracy"])
```

Model Select - MNIST

- Training sessions uses `fit` method to train model for 50 epochs.
- The `fit` method returns a `history` object that will be used to plot training curves. It uses a callback function to save the model with the lowest validation loss.

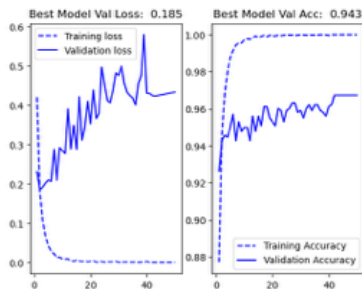
```
callbacks = [  
    keras.callbacks.ModelCheckpoint(  
        filepath="test_model.keras",  
        save_best_only = true,  
        monitor = "val_loss")]  
  
history = model.fit(ptrain_ds, epochs =50,  
                    validation_data = val_ds,  
                    callbacks = callbacks)
```

Model Selection - MNIST

```
test_model = keras.models.load_model("test_model.keras")
best_val_loss, best_val_acc = test_model.evaluate(val_ds)

import matplotlib.pyplot as plt
train_loss = history.history["loss"]
val_loss = history.history["val_loss"]
train_acc = history.history["accuracy"]
val_acc = history.history["val_accuracy"]
epochs = range(1, len(train_loss)+1)

ffigure, axis = plt.subplots(1,2)
axis[0].plot(epochs, train_loss, "b--", label = "Training loss")
axis[0].plot(epochs, val_loss, "b", label = "Validation loss")
axis[0].set_title(f"Best Model Val Loss: {best_val_loss: .3f}")
axis[0].legend()
axis[1].plot(epochs, train_acc, "b--", label = "Training Accuracy")
axis[1].plot(epochs, val_acc, "b", label = "Validation Accuracy")
axis[1].set_title(f"Best Model Val Acc: {best_val_acc: .3f}")
axis[1].legend()
```

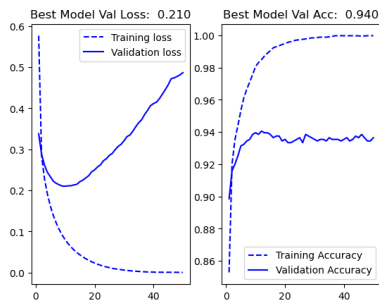


This shows that the model begins overfitting almost immediately and the best model has a validation accuracy of 94%.

Model Selection - MNIST

Since the original model overfits very early, we will make changes. One reason for overfitting is because the model is too complex. So let us try this again using a simpler model with a single hidden layer of 64 nodes.

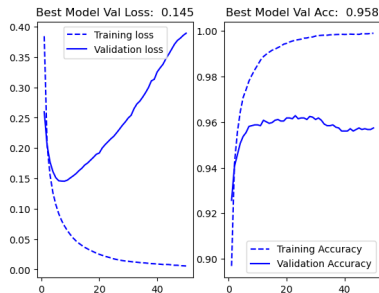
Overfitting starts after 10 epochs, but the validation accuracy doesn't change much.



Model Selection - MNIST

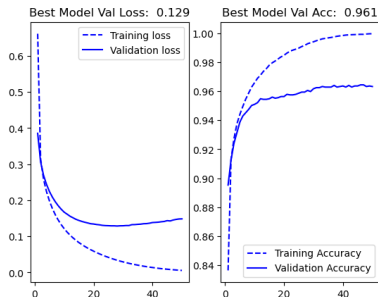
Another reason for overfitting is that the datasets are too small. We originally trained with 10,000 samples. Now let us increase this to 30,000 samples.

Model Accuracy improves to 95.8%



Model Selection - MNIST

We can change how training is done by changing batch size, reconfiguring the optimizer, or augmenting the loss with a regularization component. Let us first increase batch size from 32 to 256.

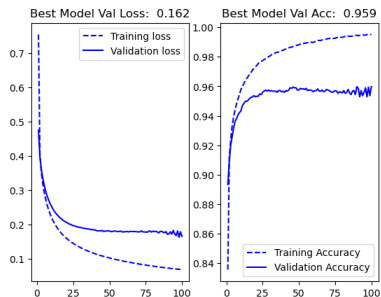


We now see a significant change in the validation loss, indicating that we are beginning to control overfitting. We also see that the accuracy has now improve to 96.1%

Model Selection - MNIST

We now try adding an L2 regularizer. This is done by declaring the regularization kernel in the layer definition.

```
inputs = keras.Input(shape=(28*28))
x = layers.Dense(64,
                 kernel_regularizer=regularizers.l2(0.001),
                 activation="relu")(inputs)
outputs = layers.Dense(10, activation="softmax")(x)
model = keras.Model(inputs=inputs, outputs=outputs)
```



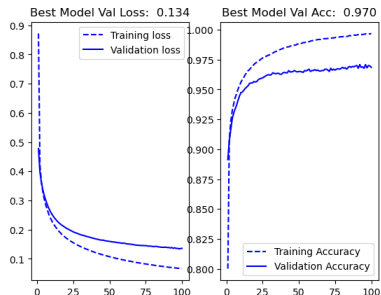
This greatly reduced the overfitting that we say before, but its impact on model accuracy was small.

Model Selection - MNIST

The last thing we try is adding a different optimizer. We switch to the ADAM optimizer with a learning rate of 0.001. This is declared in the `compile` method.

```
model.compile(  
    optimizer=tf.keras.optimizers.Adam(learning_rate=1.e-3),  
    loss = "sparse_categorical_crossentropy",  
    metrics = ["accuracy"])
```

This gave us the best performance yet 97%, and appears to give the best generalization performance.



Weight (Norm) Regularizers

- Weight regularization is a tool that is used to prevent overfitting. It usually works by augmenting the objective function (i.e. empirical risk) with an additional term that penalizes some norm of the weight vector.
- A *norm*, $\|w\|$, measures the "length" or "size" of a vector, $w \in \mathbb{R}^m$. This can be done in many ways. The basic property of any norm is that
 - Positive definite: $\|w\| \geq 0$ and $\|w\| = 0$ if and only if $w = 0$.
 - Linear Scaling: $\|\alpha w\| = |\alpha| \|w\|$
 - Triangle inequality: $\|w_1 + w_2\| \leq \|w_1\| + \|w_2\|$
- Common examples of norms are ℓ_2 , ℓ_1 , and ℓ_∞ .

$$\|w\|_{\ell_2} = \frac{1}{m} \sum_{i=1}^m |w_i|^2, \quad \|w\|_{\ell_1} = \frac{1}{m} \sum_{i=1}^m |w_i|, \quad \|w\|_{\ell_\infty} = \max_i |w_i|$$

Weight Normalization

- We regularize by augmenting the empirical risk with the ℓ_p -norm of the weight, w ,

$$\widehat{R}(w) + \lambda \|w\|_{\ell_p}$$

where $\lambda > 0$ is a parameter.

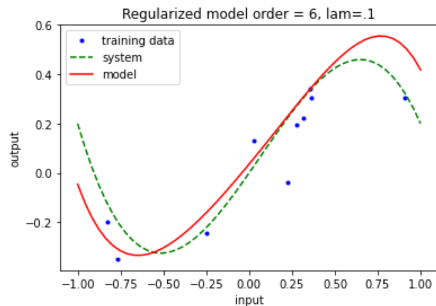
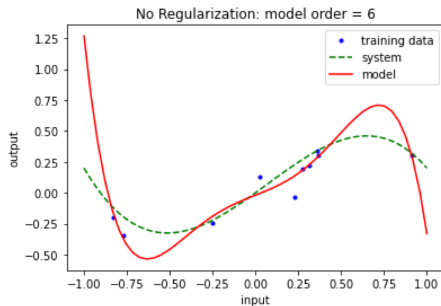
- We take the gradient of this augmented risk function and then apply backpropagation using that gradient.
- Penalizing the weight vector in this way essentially reduces our search of the model space, \mathcal{H} , to a smaller set whose effective VC dimension is smaller, thereby preventing overfitting.
- Consider a regression problem that tries fitting samples of a lower order polynomial function, $f(x)$, with a higher order polynomial.

$$J(w) = \sum_{k=1}^N \left(f(x_k) - \sum_{j=1}^M w_j x_k^j \right)^2$$

The observations for input x_k with targets

$$y_k = f(x_k) + \nu_k = x_k + 0.2x_k^2 - x_k^3 + \nu_k$$

Weight Regularization



This is clearly a very poor fit and if we look at the weight vector w , we see the weights vary over a large range from 0.02 up to 4.0. This large variation in weights is what gives rise to the large variations seen in the plot.

Norm Regularizers

- We will try to address this "overfitting" by introducing a penalty for the large weights. The easiest way of doing this is to augment $J(w)$ to

$$J_\lambda(w) = \sum_{k=1}^N \left(f(x_k) - \sum_{j=1}^M w_j x_k^j \right)^2 + \lambda \sum_{j=1}^M w_j^2$$

- To derive the optimal weight for this augmented cost, let

$$\mathbf{X} = \begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^M \\ 1 & x_2 & x_2^2 & \cdots & x_2^M \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_N & x_N^2 & \cdots & x_N^M \end{bmatrix}, \quad \mathbf{Y} = \begin{bmatrix} f(x_1) + \nu_1 \\ f(x_2) + \nu_2 \\ \vdots \\ f(x_N) + \nu_N \end{bmatrix}$$

- write our augmented objective as

$$J_\lambda(w) = \|\mathbf{X}w - \mathbf{Y}\|^2 + \lambda \|w\|^2$$

Norm Regularizers

- Taking the derivative with respect to w and setting to zero gives

$$(\mathbf{X}^T \mathbf{X} w - \mathbf{X}^T \mathbf{Y}) + \lambda w = 0$$

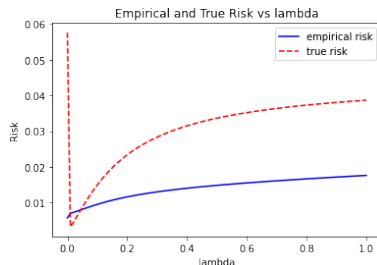
- which we rewrite as

$$[\mathbf{X}^T \mathbf{X} + \lambda \mathbf{I}] w - \mathbf{X}^T \mathbf{Y} = 0$$

- Regularized weight vector would be

$$w_r = [\mathbf{X}^T \mathbf{X} + \lambda \mathbf{I}]^{-1} \mathbf{X}^T \mathbf{Y}$$

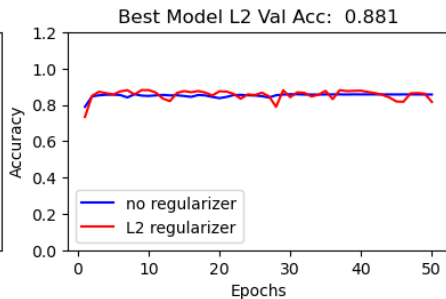
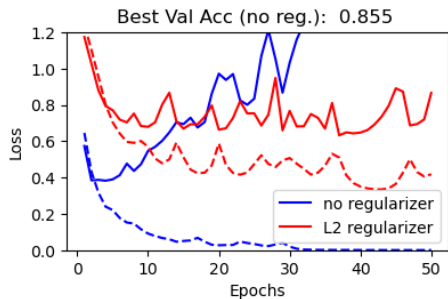
- Compute MSE as a function of the regularization parameter λ
- Shows that we have to "search" for the best λ .



- We'll evaluate our weight regularization on the IMDB database; a set of text movie reviews that have been classified as "positive" or "negative".
- The IMDB database is split into 25,000 reviews for training and 25,000 reviews for testing. Each set is split evenly between positive and negative reviews.
- The words in each review are encoded as integers, where we only encode the first 10,000 words appearing most often in the reviews.
- We therefore encode each text review as a 10000 length binary vector whose k th component is 1 if the k th most frequency work in all review appears.
- This type of encoding is called **multi-hot encoding**.
- This approach to encoding text is called a bag-of-words model in which the model correlates specific "words" appearing in the view with a positive or negative review.

Norm Regularizer -IMDB

Apply the L2 regularizer to our earlier IMDB example. In this case, we'll start with a model with 15 hidden layers of dimension 4.



Dropout Regularization

- Dropout is one of the most effective and commonly used regularization techniques for neural networks.
- Dropout is based on the idea of Bagging:
 - train several separate models
 - Have models vote on the output for an input.
 - This approach is impractical for NN.
- Dropout may be seen as a way to make bagging practical for NN. It trains an ensemble of *subnetworks* of the main model. Subnetworks are obtained by removing nodes in non-output layers during training.
- In particular, Dropout removes randomly selected non-output nodes from a layer by setting their weights to zero. This can be realized through a Keras Layer

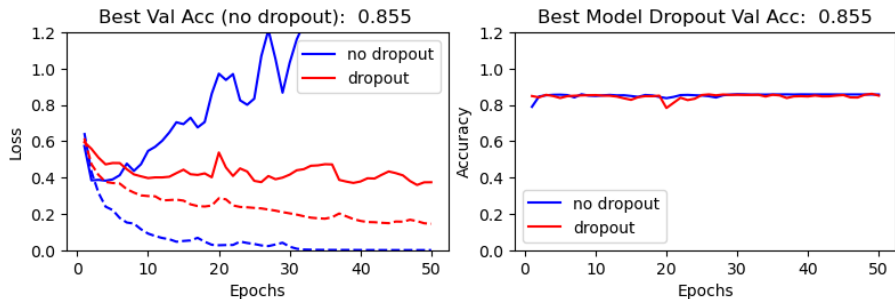
Dropout Regularization

- The dropout layer is only active during training. It is deactivated when being called through the `evaluate` or `predict` methods.
- We apply dropout (instead of ℓ_2 weight regularization) to our IMDB problem
- Recall that IMDB is a database of \pm reviews that we encode using multi-hot encoding.

```
def build_model_dropout(num_input, num_nodes, num_layers, rate):  
    inputs = keras.Input(shape = (num_input))  
    x = layers.Dense(num_nodes, activation="relu")(inputs)  
    x = layers.Dropout(rate)(x)  
    for k in range(num_layers-1):  
        x = layers.Dense(num_nodes, activation="relu")(x)  
        x = layers.Dropout(rate)(x)  
    outputs = layers.Dense(1, activation="sigmoid")(x)  
    model = keras.Model(inputs=inputs, outputs=outputs)  
    return model  
  
rate = .2  
model3 = build_model_dropout(num_input, num_nodes, num_layers, rate)
```

Dropout Regularization

We train with the same network (15 layers), but use a dropout rate of 20% on all non-output layers. The results show that dropout has a similar impact on the training curve as ℓ_2 regularization



- While gradient descent is used to train deep neural networks, this descent may be realized in many different ways.
- The **classical steepest gradient descent**

$$w_{k+1} = w_k - \gamma \nabla_w \widehat{R}_N(w)$$

where $\widehat{R}_N(w)$ is computed with respect to the entire dataset. This is "slow" and not practical for datasets that do not fit in memory

- **Stochastic Gradient Descent** (SGD) performs a gradient update for each training sample (x_k, y_k)

$$w_{k+1} = w_k - \gamma \nabla_w L(h(x_{i_k}, w_k), y_{i_k})$$

This converges quickly but is very noisy.

- **Batched SGD** a gradient update for a mini-batch, $\mathcal{D}_k = \{(x_{k_i}, y_{k_i})\}_{i=1}^{N_b}$, of samples

$$w_{k+1} = w_k - \gamma \nabla_w \left[\frac{1}{N_b} \sum_{i=1}^{N_b} L(h(x_{k_i}, w_k), y_{k_i}) \right] = w_k - \gamma \nabla_w \widehat{R}(w_k, \mathcal{D}_k)$$

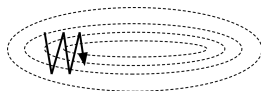
Issues with Mini-batch Optimizers

- It may be hard to find an appropriate learning rate, γ . Convergence is extremely slow if the learning rate is too small. Convergence may not occur if the learning rate is too large.
- Learning rate schedules adjust the learning rate during training by reducing the rate according to a pre-defined schedule or when the change in the objective begins to stall. These schedules and thresholds, however, are usually defined in advance and do not adapt to a dataset's characteristics.
- The same learning rate is often applied to all weights. But if the data is sparse then we may not want to update all weights with the same γ .
- If the error function is highly non-convex (common for neural networks), then it may be very difficult for the mini-batch algorithm to escape a saddle points and local minimum since the gradient is close to zero.

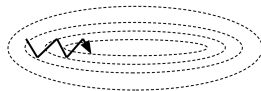
Optimizers - Nesterov Momentum

- One improvement to mini-batch SGD uses *momentum*.
- In narrow valleys of objective function, SGD may oscillate and make slow progress toward the optimal value.
- Momentum accelerates this convergence by adding a fraction, η , of the past update to the current update.
- Most useful version of this is Nesterov momentum

$$\begin{aligned}g_k &= \nabla_w \hat{R}(w_k - \eta v_{k-1}, \mathcal{D}_k) \\v_k &= \eta v_{k-1} + \gamma g_k \\w_{k+1} &= w_k - v_k\end{aligned}$$



without momentum



with momentum

Optimizers - Adagrad

- Another approach, **Adagrad**, adapts the learning rate by using a larger learning rate for weights that are updated less often.
- Adagrad is well suited to sparse data.

$$\begin{aligned}g_k &= \nabla_w \hat{R}(w_k, \mathcal{D}_k) \\n_k &= n_{k-1} + g_k \otimes g_k \\w_{k+1} &= w_k - \frac{\gamma}{\sqrt{n_k} + \epsilon} \otimes g_k\end{aligned}$$

- Note that the learning rate $\frac{\gamma}{\sqrt{n_k} + \epsilon}$ is a vector of same shape as n_k .
- Essentially, what Adagrad is doing is dividing the base learning rate γ for each weight parameter by the sum of the square of the past gradients for that parameter.

Optimizers - RMSprop

- Problem with Adagrad is that the sum dividing the learning rate is constantly increasing, so that η eventually becomes vanishingly small and learning stops.
- **RMSprop** is a variation of Adagrad that has this sum decay over time.
- The update equations are

$$\begin{aligned} \mathbf{g}_k &= \nabla_w \hat{R}(w_k, \mathcal{D}_k) \\ n_k &= \mu n_{k-1} + (1 - \mu) \mathbf{g}_k \otimes \mathbf{g}_k \\ w_{k+1} &= w_k - \frac{\gamma}{\sqrt{n_k} + \epsilon} \otimes \mathbf{g}_k \end{aligned}$$

where $\frac{\gamma}{\sqrt{n_k} + \epsilon}$ is a vector with the same shape as n_k .

- Because the component of n_k are decaying averages of the past squared gradients, the effective learning rate does not go to zero asymptotically and can "adapt" to changes in the squared gradient as we traverse the weight space.

Optimizers - Adam

- Adagrad and RMSprop are methods that adapt the learning rate, whereas the momentum based algorithms prevent the update direction from oscillating too much.
- These two techniques are combined in the adaptive moment estimation (**Adam**) algorithm
- Adam combines classical momentum (using a decaying mean instead of a decaying sum) with RMSprop.

$$\begin{aligned}g_k &= \nabla_w \widehat{R}(w_k, \mathcal{D}_k) \\m_k &= \beta_1 m_{k-1} + (1 - \beta_1) g_k \\ \widehat{m}_k &= \frac{1}{1 - \beta_1^b} m_k \\v_k &= \beta_2 v_{k-1} + (1 - \beta_2) g_k \otimes g_k \\ \widehat{v}_k &= \frac{1}{1 - \beta_2^b} v_k \\w_{k+1} &= w_k - \gamma \frac{1}{\sqrt{\widehat{v}_k + \epsilon}} \otimes \widehat{m}_k\end{aligned}$$

- Of the preceding optimizers, Adam and RMSprop are used most often in deep learning.

```
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras import optimizers

def build_model(num_input, num_nodes, num_layers):
    inputs = keras.Input(shape = (num_input))
    x = layers.Dense(num_nodes, activation="relu")(inputs)
    for k in range(num_layers-1):
        x = layers.Dense(num_nodes, activation = "relu")(x)
    outputs = layers.Dense(1, activation = "sigmoid")(x)
    model = keras.Model(inputs=inputs, outputs=outputs)
    return model

num_input = num_words
num_nodes = 4
num_layers = 1
model = build_model(num_input, num_nodes, num_layers)

learning_rate = 0.001
optimizer = optimizers.SGD(learning_rate, momentum=.1, nesterov=True)
#optimizer = optimizers.Adagrad(learning_rate)
#optimizer = optimizers.RMSprop(learning_rate)
#optimizer = optimizers.Adam(learning_rate)
model.compile(
    optimizer = optimizer,
    loss = "binary_crossentropy",
    metrics = ["accuracy"])
```

- Compare SGD with momentum, Adagrad, RMSprop and Adam on IMDB database.
- We form dataset objects using a 40% validation split with batch size of 512
- We train a simple neural network with one hidden layer with 4 nodes and ReLU activation. Output is a single densely connected node with sigmoid activation.
- We train using various optimizers assuming binary cross-entropy loss function.

Optimizers

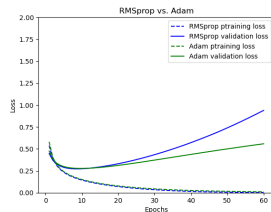
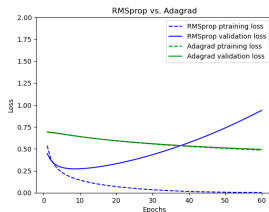
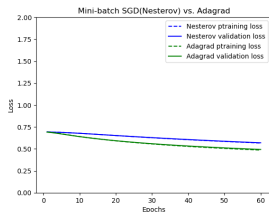


Figure: Comparison of training curves for various optimizers on a simple model for the IMDB sentiment analysis problem

Diagnosing Model Performance - Underfitting

Underfitting refers to a model that has not adequately learned the training dataset to obtain a sufficiently low training loss.

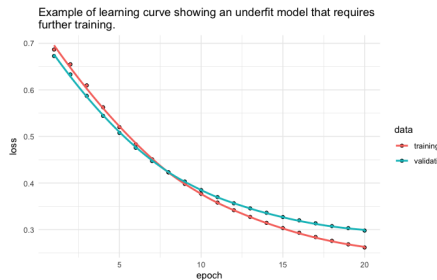
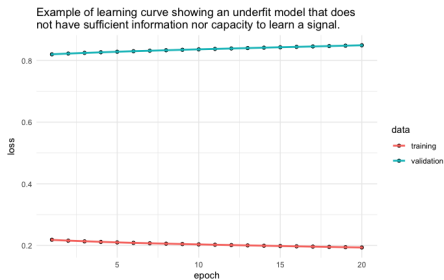


Figure: (left) learning curve showing underfitting due to insufficient data or model capacity (right) learning curve showing underfitting due to premature stopping of training.

Diagnosing Model Performance - Overfitting

Overfitting refers to a model that has learned the training dataset too well, including the statistical noise or random fluctuations in the training dataset.

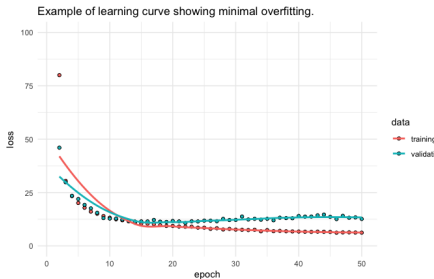
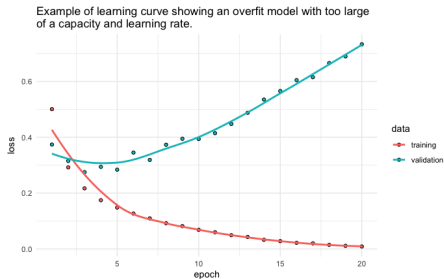


Figure: (left) learning curve showing overfitting due to excess capacity over too high learning rate (right) learning curve shows minimal overfitting.

Diagnosing Model Performance - Optimal Fit

An optimal fit, therefore, is one where 1) the training loss decreases to a stable constant value, 2) the validation loss decreases to a stable constant value, and 3) the generalization gap is small.

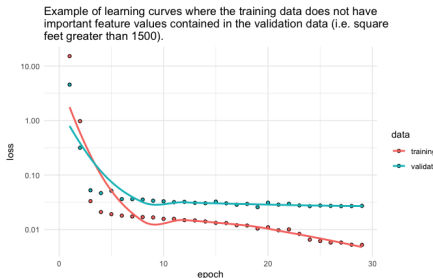
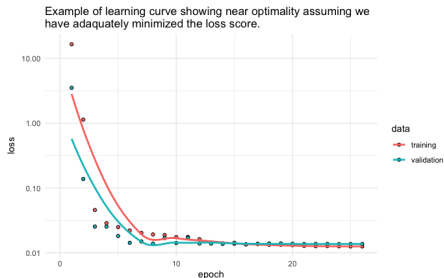


Figure: (left) learning curve showing an optimal fit (right) large generalization gap occurs because validation has features not present in training dataset.

Diagnosing Model Performance - Dataset

An unrepresentative dataset means that the dataset may not capture the statistical characteristic relative to another dataset drawn from the same domain, such as between a training and validation dataset.

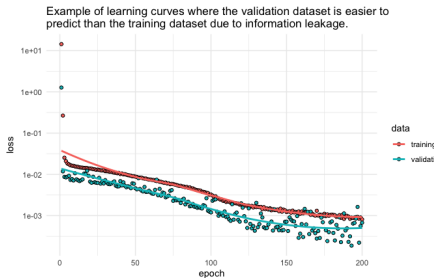
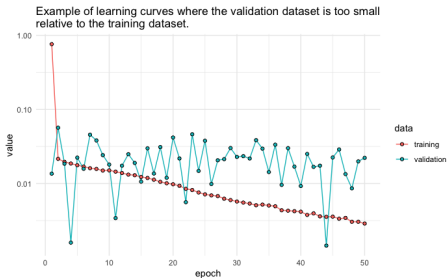


Figure: (left) validation dataset is too small relative to training data (right) training/val curves are too correlated, suggesting information leakage between the two datasets.

Summary

- Training workflow consists of four steps: problem formulation, dataset collection/curation, model selection (training), and model deployment.
- The training workflow assumes the training data set has been partitioned into a *partial* or p-training data set and a *validation* data set. The p-training data is used to minimize the model's loss function. The validation data is used to evaluate how well the trained model performed.
- The training workflow is recursive in the sense that if the performance on the validation data is not acceptable, we will go back on change the "model" or "dataset".
- After a model has been selected through this recursive process, we evaluate its performance on a test set that has been held out of training.
- The last stage of the workflow (deployment) builds an "inference" version of our trained model that can be executed on a target platform.
- Data Preparation is one of the most time consuming stages of the workflow. It involves taking the available data and placing it in a form that can be used by the model.

Summary

- Computing a performance "baseline" is an important step of data preparation. This "baseline" model usually takes the performance of the "average" class inputs as the performance level we need our trained model to beat.
- Training a model is done using mini-batch gradient descent. We usually prepare for this by loading the data samples into a Dataset object where the mini-batch sizes have already been built. This is done to speed up training.
- Training a model involves taking the p-training data set and using mini-batch gradient descent to update the model. A training epoch represents one complete pass through all of the dataset's mini-batches.
- Training and validation metrics computed at each epoch and we plot these also as a function of training epoch.
- Overfitting occurs when the validation loss begins to increase, even through the p-training loss is still decreasing (as a function of epoch).
- We control model overfitting by 1) reducing model complexity, 2) adding regularization , or 3) changing the optimizer's hyper-parameters.

Summary

- Weight decay regularization constrains how large the norm of the weights can become. We need to set a regularization parameter λ to weight the penalty of having a large weights. In general, we use either L2 or L1 norms of the weights.
- Dropout regularization is a heuristic regularization approach that works well in practice. This regularization randomly sets weights in each hidden layer to 0 during training.
- RMSprop and Adam are two of the most commonly used optimizers in ML training.
- We use learning curves we need to identify 4 conditions; underfit, overfit, optimal fit, and unrepresentative data