Deep Learning

Mairi Hallman

May 2024

Contents

1 Introduction

2 A Brief Overview of Tensors

You are likely familiar with scalars, vectors, and matrices. These can be thought of as analagous data structures in zero, one, and two-dimensions, respectively. When generalizing to N dimensions, we refer to these collectively as tensors. A scalar is a zero-order tensor, a vector is a first-order tensor, and a matrix is a second-order tensor. A third-order tensor can be visualized as a stack of matrices. A fourth-order tensor would then be a vector of third order tensors. A fifth-order tensor is a matrix of third-order tensors... and so on.

2.1 Tensor Products

Tensor additon and subtraction are self-explanatory if matrix addition and subtraction are understood. The same cannot be said for tensor products. Below is an overview of tensor products necessary for the decompositions that will be presented in the next section.

2.1.1 Outer Product $\circ$

A tensor $T^{(N)}$ can be expressed as a product of N vectors. This is called the outer product (denoted $\circ$).

$$T^{(N)} = u_1 \circ u_2 \circ \cdots \circ u_N \tag{1}$$

2.1.2 Kronecker Product $\otimes$

The Konecker product of two matrices A and B, their Kronecker product is a matrix of the products of each element in A and the entire matrix B.

$$A \otimes B = \begin{bmatrix} a_{11}B & a_{12}B & \dots & a_{1n}B \\ a_{21}B & a_{22}B & \dots & a_{2n}B \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}B & am2B & \dots & a_{mn}B \end{bmatrix} \tag{2}$$

2.1.3 Khatri-Rao Product $\odot$

The Khatri-Rao product of two matrices A and B, each with the same number of columns, is a matrix composed of the Kronecker products of the columns in matrix A and the columns in matrix B with the same indices.

$$A^{:\times n} \odot B^{:\times n} = \begin{bmatrix} a_{:,1} \otimes b_{:,1} & a_{:,2} \otimes b_{:,2} & \dots & a_{:,n} \otimes b_{:,n} \end{bmatrix} \tag{3}$$

2.1.4 Hadamard Product $*$

The Hadamard product of two matrices A and B of the same dimensions is a matrix formed of the products of the elements in A and B with the same indices.

$$A^{m \times n} * B^{m \times n} = \begin{bmatrix} a_{11}b_{11} & a_{12}b_{12} & \dots & a_{1n}b_{1n} \\ a_{21}b_{21} & a_{22}b_{22} & \dots & a_{2n}b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}b_{m1} & a_{m2}b_{m2} & \dots & a_{mn}b_{mn} \end{bmatrix} \tag{4}$$

2.2 Tensor Decompositions

2.2.1 CP Decomposition

The canonical polyadic, or CP Decomposition, decomposes a tensor into vectors.

$$T = \sum_{r-1}^{R} u_r^{(1)} \circ u_r^{(2)} \circ \dots u_r^{(R)} \tag{5}$$

2.2.2 Tucker Decomposition

The Tucker decomposition decomposes a tensor into a core tensor and factored matrices.

$$T = C \prod_{n=1}^{N} A^n \tag{6}$$

where C is the core tensor.

2.2.3 Tensor Train

The tensor train decomposition decomposes a tensor into a product of third-order tensors. It is used when a tensor is too large for the CP decomposition to be practical.

3 Why Deep Networks

Deep networks can often approximate complex functions with far fewer nodes than a shallow network. This is because the number of linear regions in the network grows exponentially with the number of layers [4].

To illustrate, consider a network with 5 layers, each with 10 nodes for a total of 50 nodes. For the sake of this example, we will assume that each layer has one input. The number of linear regions that can be learned by this network is 10^5. For a single-layer network to learn 10^5 linear regions, it would require 10^5 nodes.

4 Hidden Units and Activation Functions

Different types of hidden units serve different purposes within deep networks. Most types of hidden units perform an affine transformation on the layer's input, and then apply an activation function $g(z)$ to the transformed inputs [3]. If all activation functions within a deep network are linear, then the network reduces to a linear model [6]. While linear activation functions can be useful for reducing the number of parameters in a model , the discussion surrounding hidden units and their activation functions within a deep network typically concerns non-linear units [3].

4.1 ReLU

In modern deep networks, the default most commonly used activation function for hidden layer is the rectified linear unit, or ReLU activation function.

$$a(z) = max(0, z) \tag{7}$$

This is analagous to a linear activation function with negative inputs "turned off" [6].

Notice that this function isn't differentiable at zero. In theory, this should make ReLU useless for gradient-based learning. In practice, most deep networks don't achieve a loss of zero, so this typically isn't a problem.

Variations of ReLU typically modify the slope for negative inputs [3]. For a given input z_i, this can be generalized as

$$a(z, \alpha)_i = max(0, z_i) + \alpha_i min(0, z_i) \tag{8}$$

Several ReLU variations of this form are listed below.

Absolute Value Rectification $\alpha = -1$; commonly used for image processing [3].

Leaky ReLU α is a small positive value. Leaky ReLU is used in place of standard ReLU to prevent the "dying ReLUs" problem. Units using ReLU as an activation "die" when all of the inputs are non-positive. All negative inputs results in all zero outputs, and the weights can't be updated [2].

Parametric Leaky ReLU Leaky ReLU where *alpha* is learned by the model [2].

4.2 The Sigmoid and Hyperbolic Tangent Functions

In the early days of deep learning, two commonly used activation functions for hidden units were the sigmoid function

$$a(z) = \sigma(z) = \frac{1}{1 + e^{-z}} \tag{9}$$

and the hyperbolic tangent function

$$a(z) = tanh(z) \tag{10}$$

These activation functions are no longer recommended for hidden units because they saturate. The sigmoid function saturates at zero and one, and the hyperbolic tangent function saturates at negative one and one. Saturation is problematic because it leads to the gradient of a layer's output with respect to its input. Recall what we know about backpropagation from section 21.4.3. If the gradient is close to zero, the weights will update very slowly, if at all. Without a gradient, the model can't converge to a solution. This phenomenon is known as the vanishing gradient problem.

5 Weight Initialization

5.1 Xavier/Glorot Initialzation

5.2 Le

6 Regularization

6.1 Weight Decay

Recall ridge and LASSO regression from section 20.2.4. When generalized beyond linear models, these regularization techniques are known respectively as L^2 and L^1 regularization. In the context of neural networks, these methods are both commonly referred to as weight decay. Weight decay applies a regularization term to the objective function $J(\boldsymbol{w})$. The regularized objective function $\tilde{J}(\boldsymbol{w})$ is

$$\tilde{J}(\boldsymbol{w}) = J(\boldsymbol{w}) + \frac{\lambda}{2}\|w\|_2^2 \tag{11}$$

for L^2 weight decay, and

$$\tilde{J}(\boldsymbol{w}) = J(\boldsymbol{w}) + \lambda\|w\|_1 \tag{12}$$

for L^1 weight decay, where λ is a hyperparameter.

Tensorflow offers functionality for L^2 weight decay L^1 weight decay, and using both together [1].

```
import tensorflow as tf

n = 10 # desired number of units in layer

# dense layer with L2 weight decay
```

```
l2_regularizer=tf.keras.regularizers.L2(
    l2=0.01 # hyperparameter for l2 regularization, default
    is 0.01
)

layer = tf.keras.layers.Dense(n, regularizer=l2_regularizer)

# dense layer with L1 weight decay
l1_regularizer=tf.keras.regularizers.L1(
    l1=0.01 # hyperparameter for l2 regularization, default
    is 0.01
)

layer = tf.keras.layers.Dense(n, regularizer=l1_regularizer)

# dense layer with L2 and L1 weight decay
l1l2_regularizer=tf.keras.regularizers.L1L2(
    l1=0.01, # hyperparameter for l2 regularization, default
     is 0.01
    l2=0.01 # hyperparameter for l2 regularization, default
    is 0.01
)

layer = tf.keras.layers.Dense(n, regularizer=
    l1l2_regularizer)
```

6.2 Early Stopping

If a model is trained for too many epochs, the validation set error will reach a minumum at the optimal number of epochs, and then begin to increase as a result of overfitting. This can be prevented by implementing early stopping. When early stopping is used, a copy of the model's parameters is only saved at the end of a training iteration if the validation loss improves [3]. Training stops automatically if the validation loss hasn't decreased, or has only decreased by a very small amount, after a given number of training loops.

Early stopping can be implemented in Tensorflow as follows.

```
import tensorflow as tf

callback = tf.keras.callbacks.EarlyStopping(
    monitor=’val_loss’, # how improvement is quantified’
    min_delta=0, # decrease in monitored metric required to
    be considered an "improvement"
    patience=3, # number of epochs without improvement after
     which the model will stop training
    restore_best_weights=True, # whether to restore weights
    from the best epoch (True) or last epoch (False); should
    be set to True
    start_from_epoch=0 # how many epochs to wait before
    checking for early stop
)

model = keras.models.Sequential([
    # define your model here
        ]
)

model.compile(
    # ...
)
history = model.fit(
    # ...
    callbacks=[callback]
)
```

6.3 Dropout

Dropout is a regularization technique that randomly sets a given proportion of the models weights to zero after each gradient step. This means that for each epoch, a ”different” model is being trained [7]. This is similar to bagging, but instead of being independent, each model’s weights are a subset of the full network’s weights [3].

Dropout helps prevent overfitting by stopping units from becoming “lazy”. Consider a hockey team consisting of a few star players, many average players, and some weak players. Typically, the best players get the most playing time. The downside of this is that the others get less practice, and therefore don’t improve at the same rate. If, for each game, a certain number of players were randomly selected and told to stay hom, the rest of the team would have to adapt to play without those players. In this scenario, the team is less at risk of becoming dependent on a few superstars, and the benchwarmers get a chance to improve and contribute. In this example, the team is the model, and each player is a unit. By randomly ”dropping out” some players, the others get more practice and become better assets to the team [2].

Figure 1: Consider this meme from user @mlinterview on X. In the Marvel series, Thanos is a villain whose goal is to make half of all living organisms in the universe disappear. Those who disappear are randomly selected. While Thanon's motivations are not related to regularization, do you think this would force the remaining life forms to adapt for the better? Why or why not?

Like Thanos, we can dropout in Tensorflow (note that dropping out 50% of the weights is very excessive and not advised)[1].

```
import tensorflow as tf

my_dropout = tf.keras.layers.Dropout(
    rate=0.1 # the proportion of units to be randomly
   dropped out at each iteration
)

model = keras.models.Sequential([
    # ...
    my_dropout,
    # ...
        ]
)
```

6.4 Batch Normalization

7 Variants of Stochastic Gradient Descent

In section 21.4.3, we introduced stochastic gradient descent. Below are some common variants of SGD.

7.1 Momentum

A significant drawback of SGD is that it can be very slow to converge to an optimal solution. SGD with momentum helps address this by adding a weighted moving average of the past gradients. This allows the algorithm to converge more quickly when sequential gradients move in the same direction, and to slow down if there is a sudden change [6].

The momentum algorithm is implemented as follows. Here, $\boldsymbol{m}_t$ is the momentum, η is the learning rate, $\boldsymbol{g}_{t-1}$ is the gradient at previous output, and $\boldsymbol{\theta}_t$ is the output. β is the momentum hyperparamenter, which controls the exponential decay of the weights of past gradients. Common values of β are 0.5, 0.9, and 0.99 [6]

$$\begin{aligned} \boldsymbol{m}_t &= \beta \boldsymbol{m}_{t-1} + \boldsymbol{g}_{t-1} \\ \boldsymbol{\theta}_t &= \boldsymbol{\theta}_{t-1} - \eta \boldsymbol{m}_t \end{aligned}$$

Some resources give a different set of formulas for optimization with momentum. Note that the following are equivalent when $\boldsymbol{m}_1 = 0$.

$$\begin{aligned} \boldsymbol{m}_t &= \beta \boldsymbol{m}_{t-1} - \eta \boldsymbol{g}_t \\ \boldsymbol{\theta}_t &= \boldsymbol{\theta}_{t-1} + \boldsymbol{m}_t \end{aligned}$$

7.2 Nesterov Momentum

An improved variant of the momentum algorithm was introduced in 2013 [8]. This version, known as Nesterov momentum, uses the gradient at $\boldsymbol{\theta}_{t-1} + \beta \boldsymbol{m}_{t-1}$ instead of at $\theta t - 1$. This can be thought of as an "extrapolation" step. Instead of measuring at the previous output, we are assuming that the algorithm will continue to converge slightly ahead of this output with respect to the momentum. This allows for faster convergence.

$$\begin{aligned} \boldsymbol{g}_t &= \nabla \mathcal{L}(\boldsymbol{\theta}_t + \beta \boldsymbol{m}_t) \\ \boldsymbol{m}_{t+1} &= \beta \boldsymbol{m}_t - \eta \boldsymbol{g}_t \\ \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t + \boldsymbol{m}_{t+1} \end{aligned}$$

7.3 AdaGrad

Until now, our discussion of learning algorithms has concerned algorithms with static learning rates. The AdaGrad algorithm modifies the learning rates of the model parameters by dividing them by the square root of the Kronecker product of the vector of past gradients with itself. This speeds up convergence when the gradient is less extreme.

The algorithm is provided below. The variable s is the gradient accumulation variable, with $r_1 = 0$. η is the global learning rate, and ϵ is a very small value to prevent division by 0.

$$
\begin{aligned}
\boldsymbol{s}_t &= \boldsymbol{s}_{t-1} + \sqrt{\boldsymbol{g}_t \otimes \boldsymbol{g}_t} \\
\boldsymbol{\Delta\theta}_t &= \frac{-\eta}{\epsilon + \boldsymbol{s_t}} \otimes \boldsymbol{g} \\
\boldsymbol{\theta}_t &= \boldsymbol{\theta}_{t-1} + \boldsymbol{\Delta\theta}_t
\end{aligned}
$$

7.4 RMSProp

RMsProp is similar to AdaGrad, but with an exponentially decaying moving average gradient accumulation variable [3]. This allows the algorithm to converge more quickly as it approaches a solution.

The parameters here are the same as for AdaGrad, with the addition of β for the decay rate. A common value for β is 0.9 [6].

$$
\begin{aligned}
\boldsymbol{s}_t &= \beta\boldsymbol{s}_{t-1} + (1-\beta)\boldsymbol{g}_t \otimes \boldsymbol{g}_t \\
\boldsymbol{\Delta\theta}_t &= \frac{-\eta}{\epsilon + \boldsymbol{s_t}} \otimes \boldsymbol{g}_t \\
\boldsymbol{\theta}_t &= \boldsymbol{\theta}_{t-1} + \boldsymbol{\Delta\theta}_t
\end{aligned}
$$

7.5 Adadelta

Adadelta is a variant of RMSProp that multiplies the update by an exponetially weighted moving average of the past updates $\boldsymbol{\delta}_t = \beta\boldsymbol{\delta}_{t-1} + (1-\beta)(\boldsymbol{\delta}_t)^2$ [6] [9].

$$
\begin{aligned}
\boldsymbol{s}_t &= \beta\boldsymbol{s}_{t-1} + (1-\beta)\boldsymbol{g}_t \otimes \boldsymbol{g}_t \\
\boldsymbol{\Delta\theta}_t &= \frac{-\eta\sqrt{\boldsymbol{\delta}_{t-1}}}{\epsilon + \boldsymbol{s_t}} \otimes \boldsymbol{g}_t \\
\boldsymbol{\theta}_t &= \boldsymbol{\theta}_{t-1} + \boldsymbol{\Delta\theta}_t
\end{aligned}
$$

7.6 Adam

The adaptive moment estimation, or Adam optimizer, can be thought of as a hybrid of the momentum and RMSProp algorithms. It is good practice to correct m_t and s_t for bias to prevent bias towards smaller values [6].

$$\begin{aligned}
\boldsymbol{m}_t &= \beta_1 \boldsymbol{m}_{t-1} + (1-\beta_1)\boldsymbol{g}_{t-1} \\
\boldsymbol{s}_t &= \beta_2 \boldsymbol{s}_{t-1} + (1-\beta_3)\boldsymbol{g}_t \otimes \boldsymbol{g}_t \\
\hat{\boldsymbol{m}}_t &= \frac{\boldsymbol{m}_t}{1-\beta_1^t} \\
\hat{\boldsymbol{s}}_t &= \frac{\boldsymbol{s}_t}{1-\beta_2^t} \\
\boldsymbol{\Delta\theta}_t &= \frac{-\eta}{\epsilon + \boldsymbol{s_t}} \otimes \boldsymbol{m}_t \\
\boldsymbol{\theta}_t &= \boldsymbol{\theta}_{t-1} + \boldsymbol{\Delta\theta}_t
\end{aligned}$$

8 Convolutional Neural Networks

8.1 What Is Convolution?

Many young people who are active on social media use filters on their pictures before posting them. These filters apply different effects to the photo, such as blurring, making the image black and white, or adding cartoon-like effects. When you apply a filter to a photo, you are using convolution. Convolution is an operation that takes the aggregation of the element-wise product of a tensor of input data and a (typically smaller) tensor, called a kernel. This produces a feature map. In the context of image processing, the original image is the input data, the filter is the kernel, and the filtered image is the feature map.

This is an example of discrete convolution, which is the case that we will be primarily concerned with.

In one dimension, discrete convolution is denoted

$$\underbrace{s(i)}_{\substack{\text{feature}\\\text{map}}} = (x * y)(i) = \sum_m \underbrace{x(m)}_{\substack{\text{input}\\\text{data}}} \underbrace{y(i-m)}_{\text{kernel}} \tag{13}$$

One of the most common applications of CNNs is in digital image processing. In the case of a black and white image, the input data is a two-dimensional tensor. Discrete convolution in two dimensions can be performed as follows [3].

$$s(i,j) = (x * y)(i,j) \sum_m \sum_n x(i,j) y(i-m, j-n) \tag{14}$$

Since convolution is commutative, the following also holds.

$$s(i,j) = (y * x)(i,j) \sum_m \sum_n x(i-m, j-n) y(i,j) \tag{15}$$

Two-dimensional discrete convolution is equivalent to reversing the row and column indices of the kernel, performing element-wise multiplication, and taking the sum of the products. In the context of deep learning, the term "convolution" often refers to a similar operation called cross-correlation (equation). This is equivalent to convolution without the reversing of the matrix indices [3].

$$s(i,j) = (y * x)(i,j) \sum_{m} \sum_{n} x(i+m, j+n) y(i,j) \tag{16}$$

8.2 How Convolution is Used in Deep Neural Networks

Since one kernel can only extract one feature, each convolutional layer in a network uses multiple kernels. CNNs can be very computationally expensive to train, so we often skip over positions in the kernels to reduce training time. The width of the kernel and the size of the output can also be controlled by zero padding the input. Without zero padding, the output of each later would continue to shrink. The number of rows/columns per convolution is referred to as the stride of the convolution operation [3].

8.3 The Convolutional Layer

The convolutional layers used in a CNN have three steps [3].

1. Several parallel convolutions yield a set of linear activations.

2. Each linear activation function is run through a non-linear activation function, such as ReLU. This introduces non-linearity and gives the network more flexibility.

3. A pooling function replaces the output at a given location with a summary statistic of nearby outputs. One common pooling function is the max pooling function, which provides the maximum output within a rectangular neighbourhood. Pooling is useful when we only care about whether a feature is present and not its specific location. It is also helpful when processing different-sized inputs.

The output of each layer is then used as input to the next layer.

8.4 Image Classification Example

In this example, we will use convolutional neural networks to perform image classification on the Rock Paper Scissors dataset.

```
import tensorflow as tf
from tensorflow.keras import layers, models, Input
from tensorflow.keras.callbacks import EarlyStopping
import tensorflow_datasets as tfds
import matplotlib.pyplot as plt
import numpy as np

tf.keras.utils.set_random_seed(42) # Set random seed for
    reproducibility
```

The dataset is only partitioned into training and test sets, so we set aside 30% of the training set as a validation set.

```
(ds_train, ds_val, ds_test) = tfds.load(
    'rock_paper_scissors',
    split=['train[:70%]','train[70%:]','test'],
    shuffle_files=True,
    as_supervised=True
)
```

Next, we normalize by dividing by 255 and resize the images to 128x128. We also shuffle the training data to ensure that the model doesn't learn its order, and batch the training, validation, and test data.

```
def preprocess(image, label):
    image = tf.cast(image, tf.float32) / 255.0  # Normalize
    images
    image = tf.image.resize(image, [128, 128])  # Resize
    images
    return image, label

BATCH_SIZE = 32
SHUFFLE_BUFFER_SIZE = 1000

ds_train = ds_train.map(preprocess).shuffle(
    SHUFFLE_BUFFER_SIZE).batch(BATCH_SIZE) # shuffle so the
    model doesn't learn the order of the training data
ds_val = ds_val.map(preprocess).batch(BATCH_SIZE)
ds_test = ds_test.map(preprocess).batch(BATCH_SIZE)
```

We start with a simple network with one comvolutional layer with 16 filters and a 3x3 kernel.

```
input_shape = (128, 128, 3)

model = models.Sequential([
    layers.Input(shape=input_shape),
    layers.Conv2D(16, (3, 3), activation='relu'),  #
    Convolution step
    layers.MaxPooling2D((2, 2)),  # Pooling step
    layers.Flatten(),  # Flattening for classification
    layers.Dense(128, activation='relu'), # Dense layer
    before final classification layer
    layers.Dense(3)  # 3 classes
])

model.compile(optimizer='adam',
              loss=tf.keras.losses.
    SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy']
              )
```

Before fitting our model, we add an early stop mechanism to prevent overfitting.

```
early_stop = EarlyStopping(
    monitor='val_loss',
    min_delta=0.001,
    patience=3,
    verbose=1,
    mode='auto',
    restore_best_weights=True
)

history = model.fit(ds_train, epochs=10, validation_data=(
    ds_val), callbacks=[early_stop])
```

```
Epoch 1/10
56/56 [==============================] - 10s 85ms/step -
    loss: 1.0240 - accuracy: 0.7137 - val_loss: 0.2209 -
    val_accuracy: 0.9563
Epoch 2/10
56/56 [==============================] - 5s 61ms/step - loss
    : 0.0743 - accuracy: 0.9881 - val_loss: 0.0240 -
    val_accuracy: 0.9947
Epoch 3/10
56/56 [==============================] - 4s 37ms/step - loss
    : 0.0131 - accuracy: 0.9989 - val_loss: 0.0148 -
    val_accuracy: 0.9947
Epoch 4/10
56/56 [==============================] - 4s 37ms/step - loss
    : 0.0045 - accuracy: 1.0000 - val_loss: 0.0062 -
    val_accuracy: 0.9987
Epoch 5/10
56/56 [==============================] - 5s 58ms/step - loss
    : 0.0020 - accuracy: 1.0000 - val_loss: 0.0042 -
    val_accuracy: 1.0000
Epoch 6/10
56/56 [==============================] - 4s 38ms/step - loss
    : 0.0013 - accuracy: 1.0000 - val_loss: 0.0039 -
    val_accuracy: 1.0000
Epoch 7/10
56/56 [==============================] - 4s 38ms/step - loss
    : 0.0010 - accuracy: 1.0000 - val_loss: 0.0041 -
    val_accuracy: 0.9987
Epoch 8/10
53/56 [===========================>..] - ETA: 0s - loss:
    8.3814e-04 - accuracy: 1.0000Restoring model weights from
     the end of the best epoch: 5.
56/56 [==============================] - 5s 59ms/step - loss
    : 8.2481e-04 - accuracy: 1.0000 - val_loss: 0.0032 -
    val_accuracy: 0.9987
```

```
Epoch 8: early stopping
```

Training accuracy of 100% and validation accuracy of over 99%? If something seems too good to be true, it probably is. Let’s evaluate on the test data.

```
model.evaluate(ds_test)
```

```
12/12 [==============================] - 1s 104ms/step -
    loss: 1.3345 - accuracy: 0.7151
[1.3345420360565186, 0.7150537371635437]
```

As expected, no such luck. What happens when we add another convolutional layer?

```
model = models.Sequential([
    layers.Input(shape=input_shape),
    layers.Conv2D(16, (3, 3), activation=’relu’),  # First
    convolution step
    layers.MaxPooling2D((2, 2)),  # First pooling step
    layers.Conv2D(32, (3, 3), activation=’relu’),  # Second
    convolution step
    layers.MaxPooling2D((2, 2)),  # Second pooling step
    layers.Flatten(),  # Flattening for classification
    layers.Dense(128, activation=’relu’), # Dense layer
    before final classification layer
    layers.Dense(3)  # 3 classes
])

model.compile(optimizer=’adam’,
              loss=tf.keras.losses.
    SparseCategoricalCrossentropy(from_logits=True),
              metrics=[’accuracy’])

history = model.fit(ds_train, epochs=10, validation_data=(
    ds_val),callbacks=[early_stop])
```

```
 Epoch 1/10
 56/56 [==============================] - 6s 43ms/step -
    loss: 1.4043 - accuracy: 0.5890 - val_loss: 0.4816 -
    val_accuracy: 0.9206
 Epoch 2/10
 56/56 [==============================] - 5s 54ms/step -
    loss: 0.2150 - accuracy: 0.9552 - val_loss: 0.0609 -
    val_accuracy: 0.9881
 Epoch 3/10
 56/56 [==============================] - 4s 38ms/step -
    loss: 0.0441 - accuracy: 0.9909 - val_loss: 0.0182 -
    val_accuracy: 0.9987
 Epoch 4/10
 56/56 [==============================] - 4s 39ms/step -
    loss: 0.0143 - accuracy: 0.9977 - val_loss: 0.0076 -
    val_accuracy: 0.9987
 Epoch 5/10
 56/56 [==============================] - 5s 60ms/step -
    loss: 0.0053 - accuracy: 0.9994 - val_loss: 0.0030 -
    val_accuracy: 1.0000
 Epoch 6/10
 56/56 [==============================] - 4s 41ms/step -
    loss: 0.0022 - accuracy: 1.0000 - val_loss: 0.0019 -
    val_accuracy: 1.0000
 Epoch 7/10
 56/56 [==============================] - 4s 39ms/step -
    loss: 0.0016 - accuracy: 1.0000 - val_loss: 0.0013 -
    val_accuracy: 1.0000
 Epoch 8/10
 56/56 [==============================] - 5s 60ms/step -
    loss: 9.5238e-04 - accuracy: 1.0000 - val_loss: 0.0011 -
    val_accuracy: 1.0000
 Epoch 9/10
 56/56 [==============================] - 4s 38ms/step -
    loss: 6.9435e-04 - accuracy: 1.0000 - val_loss: 8.4804e
    -04 - val_accuracy: 1.0000
 Epoch 10/10
 56/56 [==============================] - 4s 38ms/step -
    loss: 5.3139e-04 - accuracy: 1.0000 - val_loss: 7.7833e
    -04 - val_accuracy: 1.0000

\lstinputlisting[firstline=100,lastline=100]{dl-scripts/cnn-
    ex1-colab.py}

\begin{lstlisting}[style=output]
12/12 [==============================] - 0s 16ms/step - loss
    : 1.0663 - accuracy: 0.7634
[1.0662583112716675, 0.7634408473968506]
```

An improvement, but still not amazing. Would a third convolutional layer help?

```
model = models.Sequential([
    layers.Input(shape=input_shape),
    layers.Conv2D(16, (3, 3), activation='relu'),  # First
    convolution step
    layers.MaxPooling2D((2, 2)),  # First pooling step
    layers.Conv2D(32, (3, 3), activation='relu'),  # Second
    convolution step
    layers.MaxPooling2D((2, 2)),  # Second pooling step
    layers.Conv2D(64, (3, 3), activation='relu'),  # Third
    convolution step
    layers.MaxPooling2D((2, 2)),  # Third pooling step
    layers.Flatten(),  # Flattening for classification
    layers.Dense(128, activation='relu'), # Dense layer
    before final classification layer
    layers.Dense(3)  # 3 classes
])

model.compile(optimizer='adam',
              loss=tf.keras.losses.
    SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])

history = model.fit(ds_train, epochs=10, validation_data=(
    ds_val),callbacks=[early_stop])
```

```
 Epoch 1/10
 56/56 [==============================] - 7s 44ms/step -
    loss: 0.6999 - accuracy: 0.7035 - val_loss: 0.1999 -
    val_accuracy: 0.9101
 Epoch 2/10
 56/56 [==============================] - 4s 39ms/step -
    loss: 0.0600 - accuracy: 0.9864 - val_loss: 0.0188 -
    val_accuracy: 0.9934
 Epoch 3/10
 56/56 [==============================] - 5s 53ms/step -
    loss: 0.0240 - accuracy: 0.9949 - val_loss: 0.0071 -
    val_accuracy: 1.0000
 Epoch 4/10
 56/56 [==============================] - 4s 39ms/step -
    loss: 0.0022 - accuracy: 1.0000 - val_loss: 0.0011 -
    val_accuracy: 1.0000
 Epoch 5/10
 56/56 [==============================] - 4s 41ms/step -
    loss: 7.2477e-04 - accuracy: 1.0000 - val_loss: 7.4349e
    -04 - val_accuracy: 1.0000
 Epoch 6/10
 56/56 [==============================] - 5s 63ms/step -
    loss: 4.0359e-04 - accuracy: 1.0000 - val_loss: 3.2628e
    -04 - val_accuracy: 1.0000
 Epoch 7/10
 52/56 [==========================>...] - ETA: 0s - loss:
    2.2042e-04 - accuracy: 1.0000Restoring model weights from
     the end of the best epoch: 4.
 56/56 [==============================] - 4s 39ms/step -
    loss: 2.1410e-04 - accuracy: 1.0000 - val_loss: 3.2290e
    -04 - val_accuracy: 1.0000
 Epoch 7: early stopping
```

```
model.evaluate(ds_test)
```

```
12/12 [==============================] - 0s 10ms/step - loss
    : 1.1395 - accuracy: 0.7930
```

```
[1.1395031213760376, 0.7930107712745667]
```

A slight improvement. What about a fourth layer?

```
model = models.Sequential([
    layers.Input(shape=input_shape),
    layers.Conv2D(16, (3, 3), activation=’relu’),  # First
   convolution step
    layers.MaxPooling2D((2, 2)),  # First pooling step
    layers.Conv2D(32, (3, 3), activation=’relu’),  # Second
   convolution step
    layers.MaxPooling2D((2, 2)),  # Second pooling step
    layers.Conv2D(64, (3, 3), activation=’relu’),  # Third
   convolution step
    layers.MaxPooling2D((2, 2)),  # Third pooling step
    layers.Conv2D(128, (3, 3), activation=’relu’),  # Fourth
    convolution step
    layers.MaxPooling2D((2, 2)),  # Fourth pooling step
    layers.Flatten(),  # Flattening for classification
    layers.Dense(128, activation=’relu’), # Dense layer
   before final classification layer
    layers.Dense(3)  # 3 classes
])

model.compile(optimizer=’adam’,
              loss=tf.keras.losses.
   SparseCategoricalCrossentropy(from_logits=True),
              metrics=[’accuracy’])

history = model.fit(ds_train, epochs=10, validation_data=(
   ds_val),callbacks=[early_stop])
```

```
Epoch 1/10
56/56 [==============================] - 6s 46ms/step - loss
   : 0.7071 - accuracy: 0.6604 - val_loss: 0.1421 -
   val_accuracy: 0.9563
Epoch 2/10
56/56 [==============================] - 5s 40ms/step - loss
   : 0.0583 - accuracy: 0.9853 - val_loss: 0.0080 -
   val_accuracy: 0.9987
Epoch 3/10
56/56 [==============================] - 4s 39ms/step - loss
   : 0.0057 - accuracy: 0.9983 - val_loss: 0.0020 -
   val_accuracy: 1.0000
Epoch 4/10
56/56 [==============================] - 4s 48ms/step - loss
   : 0.0026 - accuracy: 0.9989 - val_loss: 0.0017 -
   val_accuracy: 1.0000
Epoch 5/10
56/56 [==============================] - 5s 40ms/step - loss
   : 0.0038 - accuracy: 0.9994 - val_loss: 0.0136 -
```

```
    val_accuracy: 0.9921
Epoch 6/10
56/56 [==============================] - 4s 41ms/step - loss
    : 0.0018 - accuracy: 0.9994 - val_loss: 2.4979e-04 -
    val_accuracy: 1.0000
Epoch 7/10
56/56 [==============================] - 4s 43ms/step - loss
    : 1.2425e-04 - accuracy: 1.0000 - val_loss: 1.3199e-04 -
    val_accuracy: 1.0000
Epoch 8/10
56/56 [==============================] - 5s 39ms/step - loss
    : 9.3137e-05 - accuracy: 1.0000 - val_loss: 9.6794e-05 -
    val_accuracy: 1.0000
Epoch 9/10
54/56 [===========================>..] - ETA: 0s - loss:
    7.5662e-05 - accuracy: 1.0000Restoring model weights from
     the end of the best epoch: 6.
56/56 [==============================] - 4s 39ms/step - loss
    : 7.4308e-05 - accuracy: 1.0000 - val_loss: 7.7891e-05 -
    val_accuracy: 1.0000
Epoch 9: early stopping
```

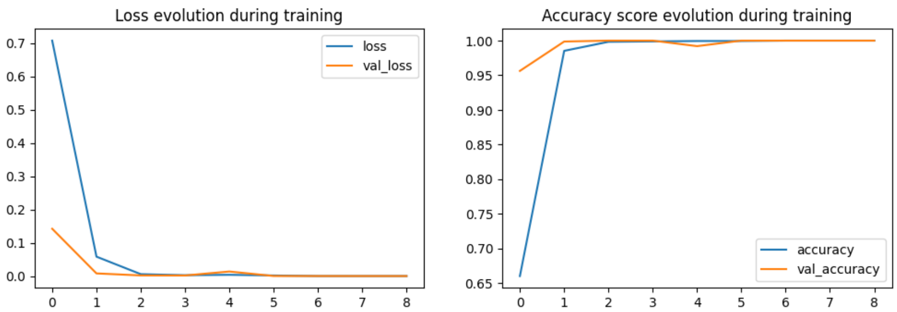
```
model.evaluate(ds_test)
```

```
12/12 [==============================] - 0s 10ms/step - loss
    : 0.1825 - accuracy: 0.9409
[0.1824917197227478, 0.9408602118492126]
```

Now that we have reached the accuracy threshold, let's plot the learning curves.

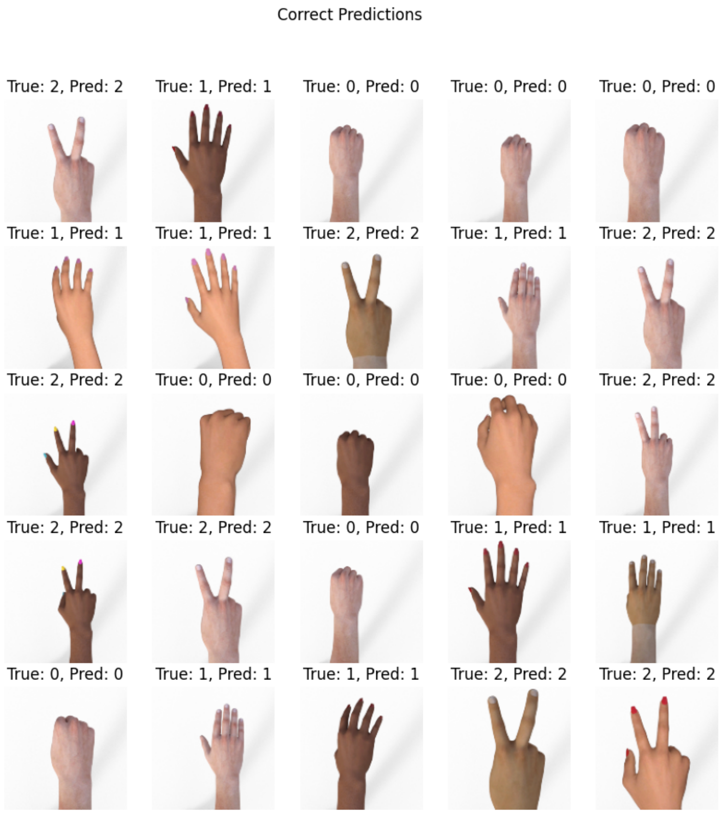
```
def plot_learning_curves(history):
    plt.figure(figsize=(12, 8))

    plt.subplot(2, 2, 1)
    plt.plot(history.history['loss'], label='loss')
    plt.plot(history.history['val_loss'], label='val_loss')
    plt.title('Loss evolution during training')
    plt.legend()

    plt.subplot(2, 2, 2)
    plt.plot(history.history['accuracy'], label='accuracy')
    plt.plot(history.history['val_accuracy'], label='
    val_accuracy')
    plt.title('Accuracy score evolution during training')
    plt.legend();

plot_learning_curves(history)
```

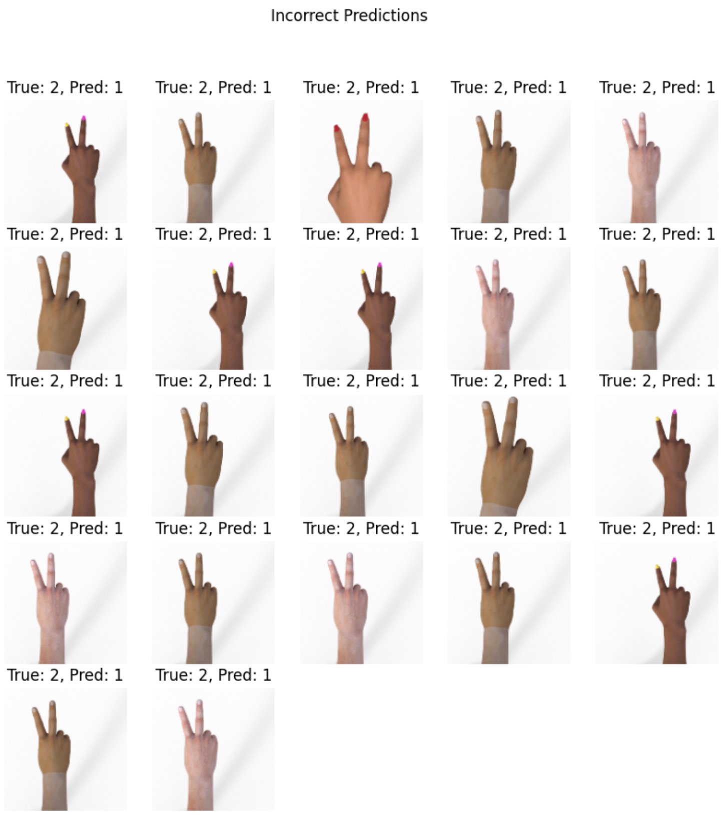
```
# Make predictions on the test set
test_images, test_labels = [], []
```



```
for images, labels in ds_test:
    test_images.append(images.numpy())
    test_labels.append(labels.numpy())
test_images = np.concatenate(test_images)
test_labels = np.concatenate(test_labels)

predictions = model.predict(test_images)
predicted_labels = np.argmax(predictions, axis=1)

# Identify correct and incorrect predictions
correct_predictions = predicted_labels == test_labels
incorrect_predictions = predicted_labels != test_labels

# Get indices for correct and incorrect predictions
correct_indices = np.where(correct_predictions)[0]
incorrect_indices = np.where(incorrect_predictions)[0]

# Function to plot images with their predictions and true
    labels
def plot_predictions(images, true_labels, predicted_labels,
    indices, title):
    plt.figure(figsize=(10, 10))
    for i, index in enumerate(indices[:25]):  # Plot the
    first 25 images
        plt.subplot(5, 5, i + 1)
        plt.imshow(images[index])
        plt.title(f"True: {true_labels[index]}, Pred: {
    predicted_labels[index]}")
        plt.axis('off')
    plt.suptitle(title)
    plt.show()

# Plot correct predictions
plot_predictions(test_images, test_labels, predicted_labels,
     correct_indices, title="Correct Predictions")

# Plot incorrect predictions
```

```
plot_predictions(test_images, test_labels, predicted_labels,
    incorrect_indices, title="Incorrect Predictions")
```





Look at the images that were classified incorrectly. It seems that the model's most significant weakness is misclassifying scissors as paper. Do you notice anything about these scissor images compared to the scissor and paper images in the correct predictions? How could the model potentially be improved?

9 Recurrent Neural Networks

Until now, we have been dealing with deep networks that only flow in one direction: forward (hence the term "feed-forward" networks). These networks are limited in that they can only take in and ouput sequences of a predetermined size. Recurrent neural networks can handle inputs of varying lengths. They are used for sequential inputs, such as text and time series data.

In a recurrent neural network, the output of a node is used as input to the next layer, as in a feedforward network, but may also be input to the same node at the next time step t [2]. More formally, node's hidden state h_t is a function of its new inputs and its output at h_{t-1} [3]. Depending on the type of RNN, the hidden state may or may not be equal to the output at the previous time step.

$$\boldsymbol{h}_t = f(\boldsymbol{h}_{t-1}, \boldsymbol{x}_t, \theta)$$

The four main types of recurrent neural networks are sequence-to-sequence, sequence-to-vector, vector-to-sequence, and encoder-decoder.

9.1 Vector-to-Sequence RNNs

Vector-to-sequence RNNs are used to generate sequences of variable lengths from vectors of a fixed length. At each time step, we take a sample of the outputs from h_t, and use these as inputs for the next time step to get h_{t+1}. This can be denoted by the conditional generative model, where T is the length of the input sequence.

$$\begin{aligned} p(\boldsymbol{y}_{1:T}|\boldsymbol{x}) &= \sum_{\boldsymbol{h}_{1:T}} p(\boldsymbol{y}_{1:T}, \boldsymbol{h}_{1:T}|\boldsymbol{x}) \\ &= \sum_{\boldsymbol{h}_{1:T}} \prod_{t=1}^{T} p(\boldsymbol{y}_t|\boldsymbol{h}_t) p(\boldsymbol{h}_t|\boldsymbol{h}_{t-1}, \boldsymbol{y}_{t-1}, \boldsymbol{x}) \end{aligned}$$

9.2 Sequence-to-Vector RNNs

A sequence-to-vector RNN takes in a sequence of variable length, and outputs a vector of fixed length. Rather than producing an output at each time step, new input at each time step, but ignores the outputs until the last time step. This is most commonly used for classification of input sequences of varying lengths. For example, a sequence-to-vector model could be used to emails as legitimate or spam. We can represent this RNN as a conditional generative model [6].

$$p(\boldsymbol{y}|\boldsymbol{x}_{1:T}) = \text{Categorical}(\boldsymbol{y}_t|\text{softmax}(\boldsymbol{W}\boldsymbol{h}_T))$$

9.3 Sequence-to-sequence RNNs

Next, we will consider RNNs with variable input and output lengths. Here, we will assume that the input and output sequences are the same size. In this case, the conditional generative model is defined as follows [6].

$$p(y_{1:T}|x_{1:T}) = \sum_{h_{1:T}} \prod_{t=1}^{T} p(y_t|h_t) I(h_t = f(h_{t-1}, x_t))$$

The inital hidden state is $h_1 = f(h_0, x_1) = f_0(x_1)$ [6].

9.4 Encoder-Decoder Models

What if we want a sequence-to-sequence model where the output and input sequences aren't necessarily the same size? In this case, we use an encoder-decoder network. Here, the encoder is a sequence-to-vector RNN, and the decoder is a vector-to-sequence RNN. The decoder takes the encoder's output as its input to generate a sequence.

9.5 Long-term memory

Due to vanishing gradients, RNNs have a tendency to "forget" information that is too far into the past. To overcome this, we can use specialized recurrent units. These units allow the network to learn which information to remember long-term, and which to discard [2].

9.5.1 Long SHort-Term Memory (LSTM) Units

LSTM cells add additional information to the hidden state via a memory cell [6]. We can think of the hidden state as short-term memory, and the memory cell's state as long-term memory. Which information is stored in the memory cell is controlled by an input gate, a forget gate, and an output gate. Below is an overview of how these gates interact with the memory cell to control the unit's long-term memory and update the hidden state.

$$
\begin{aligned}
\boldsymbol{I}_t &= \sigma(\boldsymbol{X}_t \boldsymbol{W}_{xi} + \boldsymbol{H}_{t-1} W_{hi} + \boldsymbol{b}_i) \\
\boldsymbol{F}_t &= \sigma(\boldsymbol{X}_t \boldsymbol{W}_{xt} + \boldsymbol{H}_{t-1} W_{hf} + \boldsymbol{b}_f) \\
\boldsymbol{O}_t &= \sigma(\boldsymbol{X}_t \boldsymbol{W}_{xo} + \boldsymbol{H}_{t-1} W_{ho} + \boldsymbol{b}_o) \\
\tilde{\boldsymbol{C}}_t &= tanh(\boldsymbol{X}_t \boldsymbol{W}_{xc} + \boldsymbol{H}_{t-1} \boldsymbol{W}_{hc} + \boldsymbol{b}_c) \\
\boldsymbol{C}_t &= \boldsymbol{F}_t * \boldsymbol{C}_{t-1} + \boldsymbol{I}_t * \boldsymbol{tildeC}_t && \text{final memory cell computed based on previous memory cell and canc} \\
\boldsymbol{H}_t &= \boldsymbol{O}_t * tanh(\boldsymbol{C}_t)
\end{aligned}
$$

9.5.2 Gated Recurrent Units (GRUs)

GRUs are a simplified variation of LSTM units that elimimate the need for a memory cell. The input and forget gates are replaced with one update gate $\boldsymbol{Z}_t$. A reset gate is added to determine which components of the previous state will be used to get the next state [2], and the output gate is removed entirely [6].

$$
\begin{aligned}
\boldsymbol{R_t} &= \sigma(\boldsymbol{X}_t \boldsymbol{W}_{xr} + \boldsymbol{H}_{t-1} \boldsymbol{W}_{hr} + \boldsymbol{b}_r) && resetgate \\
\boldsymbol{Z_t} &= \sigma(\boldsymbol{X}_t \boldsymbol{W}_{xz} + \boldsymbol{H_{t-1}} \boldsymbol{W}_{ht} + \boldsymbol{b}_z) && updategate \\
\tilde{\boldsymbol{H}}_{\boldsymbol{t}} &= tanh(\boldsymbol{H}_t - \boldsymbol{W}_{xh} + (\boldsymbol{R}_t * \boldsymbol{H_{t-1}}) \boldsymbol{W}_{hh} + \boldsymbol{b}_h) && candidatehiddenstate \\
\boldsymbol{H}_t &= \boldsymbol{Z}_t * \boldsymbol{H}_{t-1} + (1 - \boldsymbol{Z}_t) * \boldsymbol{H}_t && finalhiddenstate
\end{aligned}
$$

10 Specialized Architectures

This section presents a very brief overview of several specialized network architectures. Optional further reading is incicated at the end of each passage.

10.1 Generative Adversarial Neural Networks

Young children typically aren't very good liars. As people get older, they acquire more experience getting caught in bad lies and getting away with good ones. This experience should, in theory, make them better liars.

This scenario illustrates the idea behind generative adversarial networks, or GANs. GANs consist of two networks; a generator and a discriminator. The generator outputs samples $\boldsymbol{x} = g(\boldsymbol{z};\boldsymbol{\theta_{(g)}})$. The generated data and real data are input to the discriminator, which must then assign each point a probability $d(X; theta^{(d)})$. This corresponds to the probability that x is a "real" data point (not produced by the generator). The payoff of the discriminator [3] (its ability to differentiate between real and fake data) can be denoted

$$v(\boldsymbol{theta}^{(g)},\boldsymbol{theta}^{(d)} = \mathbf{E}_{x\ p_{\text{data}}} log[d(x)] + mathbf E_{x\ p_{\text{model}}} log[1-d(x)])$$

and the optimal generating function [3] is

$$g* = \underset{g}{argmin}(\underset{d}{max} v(g,d))$$

The optimal generating function minimizes the payoff of the discriminator ($\underset{g}{argmin}$), which itself is trying to maximize its payoff ($undersetdmaxv(g,d)$).

Readers seeking a more in-depth treatment of GANs are invited to consult section 20.10.4 of *Deep Learning* by Goodfellow et al. and chapter 26 of *Probabilistic Machine Learning: Advanced Topics* by Murphy.

10.2 Autoencoders and Variational Autoencoders

Autoencoders are a type of network used to copy an input to a reduced, userful output. In a traditional determininistic autoencoder, the encoder function f transforms the input to a code $h = f(x)$. The reconstruction function g outputs a reconstruction of the input $r = g(x)$. Typically, the reconstruction will be constrained in some way to ensure that the reconstruction has reduced dimensions with respect to the input [3].

Readers wishing to learn more about autoencoders are invited to consult chapter 14 of *Deep Learning* by

Variational autoenoders are the probabilistic analogue of traditional autoencoders. Rather than outputting a single point, the encoder outputs a probability distribution $q(z|x) = N(z|\mu, \sum$. This is the posterior distribution of the latent space given an input x. The mean and covariance matrix are learned by the network. A sample z is taken from this distribution and input to a generator g to produce an output $g(x)$.

The above is a very brief discussion of VAEs. For a more complete explanation, including reparameterization of the samples and derivation of the objective function, please see section 20.10.3 of *Deep Learning* by Goodfellow et al. and chapter 21 of *Probabilistic Machine Learning: Advanced Topics* by Murphy.

References

[1] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems.* Software available from tensorflow.org. 2015. URL: `https://www.tensorflow.org/`.

[2] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, Third Edition.* O'Reilly Media, Inc, 2022.

[3] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* MIT Press, 2016. URL: `http://www.deeplearningbook.org`.

[4] Guido Montúfar et al. *On the Number of Linear Regions of Deep Neural Networks.* 2014. arXiv: `1402.1869 [id='stat.ML' full_name='Machine Learning' is_active=True alt_name=None in_archive='stat' is_general=False description='Covers machine learning papers (supervised, unsupervised, semi-supervised learning, graphical models, reinforcement learning, bandits, high dimensional inference, etc.) with a statistical or theoretical grounding']`.

[5] Kevin P. Murphy. *Probabilistic Machine Learning: Advanced Topics.* MIT Press, 2023. URL: `http://probml.github.io/book2`.

[6] Kevin P. Murphy. *Probabilistic Machine Learning: An Introduction.* MIT Press, 2022. URL: `probml.ai`.

[7] Nitish Srivastava et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *Journal of Machine Learning Research* 15.56 (2014), pp. 1929–1958. URL: `http://jmlr.org/papers/v15/srivastava14a.html`.

[8] Ilya Sutskever et al. "On the importance of initialization and momentum in deep learning". In: *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28.* ICML'13. Atlanta, GA, USA: JMLR.org, 2013, III–1139–III–1147.

[9] Matthew D. Zeiler. *ADADELTA: An Adaptive Learning Rate Method.* 2012. arXiv: `1212.5701 [cs.LG]`. URL: `https://arxiv.org/abs/1212.5701`.