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 and Matrix Products

Below is an overview of tensor and matrix products necessary for the decompositions that will be presented in the next section.

2.1.1 Outer Product $\circ$

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

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

2.1.2 Kronecker Product $\otimes$

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

$$\boldsymbol{A} \otimes \boldsymbol{B} = \begin{bmatrix} a_{11}\boldsymbol{B} & a_{12}\boldsymbol{B} & \dots & a_{1n}\boldsymbol{B} \\ a_{21}\boldsymbol{B} & a_{22}\boldsymbol{B} & \dots & a_{2n}\boldsymbol{B} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}\boldsymbol{B} & am2\boldsymbol{B} & \dots & a_{mn}\boldsymbol{B} \end{bmatrix}$$

2.1.3 Khatri-Rao Product $\odot$

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

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

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.

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

2.2 Tensor Decompositions

2.2.1 CP Decomposition

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

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

2.2.2 Tucker Decomposition

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

$$\mathsf{T} = C \prod_{n=1}^{N} A^n \tag{3}$$

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.

3.1 Getting Started

To help you get comfortable with neural network architecture, we will use Pytorch and Lightning code a network to complete a task that you are already very familiar with: linear regression. First, import dependencies and set a seed for reproductibility.

```
import lightning as L
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset

torch.manual_seed(0)
```

Generate some dummy data according to the relationship $y = 3x + 1$.

```
x = torch.randn(100, 1) * 10 # sample x from a standard
    normal distribution multiplied by 10
y = 3 * x + 1 + torch.randn(100, 1) # compute y, add noise
dataset = TensorDataset(x, y) # initialize tensor dataset
train_loader = DataLoader(dataset, batch_size=10, shuffle=
    True) # initialize data loader
```

Once the data loader has been initialized, define a class for our model.

```
class LinearRegressionModel(L.LightningModule):
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(1, 1)

    # forward pass
    def forward(self, x):
        return self.linear(x)

    # training step; later examples will also include a
    validation set.
    def training_step(self, batch, batch_idx):
        x, y = batch
        y_pred = self(x)
        loss = nn.functional.mse_loss(y_pred, y)
        self.log('train_loss', loss)
        return loss

    # stochastic gradient descent for optimization
    def configure_optimizers(self):
        return optim.SGD(self.parameters(), lr=0.01)
```

Now the fun part - training!

```
model = LinearRegressionModel()
trainer = L.Trainer(max_epochs=100)
trainer.fit(model, train_loader)
```

```
INFO: GPU available: False, used: False
INFO:lightning.pytorch.utilities.rank_zero:GPU available:
    False, used: False
INFO: TPU available: False, using: 0 TPU cores
INFO:lightning.pytorch.utilities.rank_zero:TPU available:
    False, using: 0 TPU cores
INFO: HPU available: False, using: 0 HPUs
INFO:lightning.pytorch.utilities.rank_zero:HPU available:
    False, using: 0 HPUs
INFO:
  | Name   | Type   | Params | Mode
-----------------------------------------
0 | linear | Linear | 2      | train
-----------------------------------------
2         Trainable params
0         Non-trainable params
2         Total params
0.000     Total estimated model params size (MB)
1         Modules in train mode
0         Modules in eval mode
INFO:lightning.pytorch.callbacks.model_summary:
  | Name   | Type   | Params | Mode
-----------------------------------------
0 | linear | Linear | 2      | train
-----------------------------------------
2         Trainable params
0         Non-trainable params
2         Total params
0.000     Total estimated model params size (MB)
1         Modules in train mode
0         Modules in eval mode
/usr/local/lib/python3.10/dist-packages/lightning/pytorch/
    loops/fit_loop.py:298: The number of training batches
    (10) is smaller than the logging interval Trainer(
    log_every_n_steps=50). Set a lower value for
    log_every_n_steps if you want to see logs for the
    training epoch.
Epoch 99: 100% [==============================] 10/10
    [00:00<00:00, 50.34it/s, v_num=1]
INFO: `Trainer.fit` stopped: `max_epochs=100` reached.
INFO:lightning.pytorch.utilities.rank_zero:`Trainer.fit`
    stopped: `max_epochs=100` reached.
```

Let's see how our model did.

```
slope, intercept = model.linear.weight.item(), model.linear.
    bias.item()
print(f'Trained Model Parameters: Slope: {slope:.4f},
    Intercept: {intercept:.4f}')
```

```
Trained Model Parameters: Slope: 2.9757, Intercept: 0.9874
```

That's pretty good! In the coming sections, we will explore more advanced techniques for training deeper networks, as well as different specialized network arcitectures.

4 Activation Functions and Weight Initialization for Hidden Layers

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 layers is the rectified linear unit, or ReLU activation function.

$$a(z) = max(0, z)$$

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

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

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].

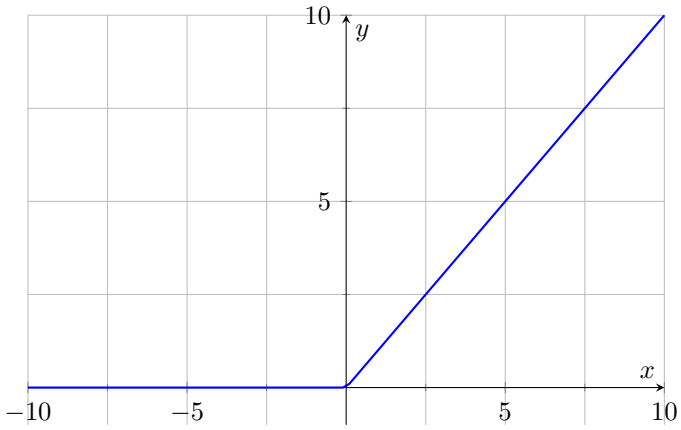


Figure 1: The ReLU activation function.

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

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

and the hyperbolic tangent function

$$a(z) = tanh(z)$$

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.

4.3 Weight Initialization

As mentioned in chapter 21, before a deep network can be trained, its weights have to be randomly initialized. Proper initialization of weights is crucial for fast convergence and preventing exploding gradients. Random weights are often sampled from a normal distribution. If the variance of this distribution is fixed for all layers, gradients may explode during training [6]. It is therefore reccommended to use one of the weight initialization strategies below.

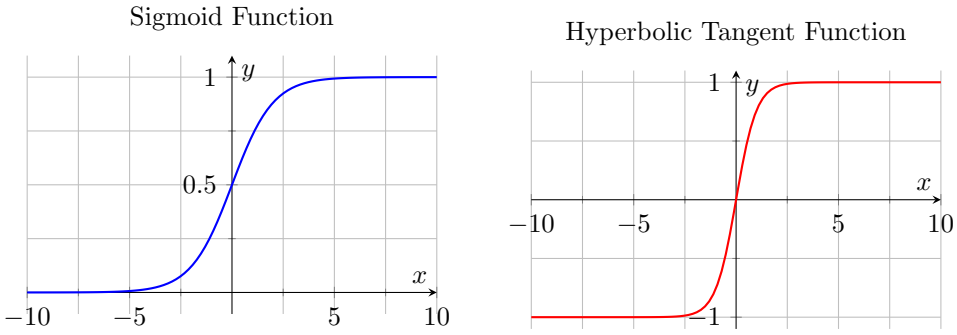


Figure 2: The sigmoid and hyperbolic tangent functions.

Glorot Initialization , also know as Xavier initialization, sets the variance of the sampling distribution for each layer to $\sigma^2 = \frac{2}{n_{in}+n_{out}}$, where n_in is the number of input connections to a unit in the layer, and n_out is the number of output connections. This is the preferred initialization scheme for sigmoid activation [6] and, by extension, hyperbolic tangent activation.

LeCunn Initialization uses a variance of $\sigma^2 = \frac{1}{n_{in}}$, and is identical to Glorot initialization with an equal number of input and output connections [6].

He Initialization uses a variance of $\sigma^2 = \frac{2}{n_{in}}$, and is the preferred initialization scheme for ReLU activation.

5 Regularization

5.1 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

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

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

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

5.2 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.

5.3 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 [9]. 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 3: 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?

Please note that Thanos was unpopular for a reason; dropping out 50% of the weights is very excessive and not advised [1].

6 Variants of Stochastic Gradient Descent

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

6.1 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 $\boldsymbol{m}_1 = 0$.

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

6.2 Nesterov Momentum

An improved variant of the momentum algorithm was introduced in 2013 [10]. 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.

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

6.3 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.

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

6.4 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].

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

6.5 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] [11].

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

6.6 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].

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

7 Convolutional Neural Networks

7.1 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

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

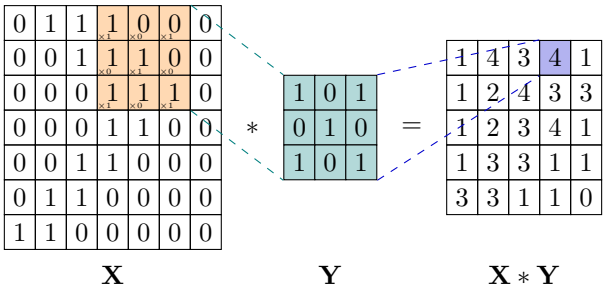


Figure 4: The convolution of an input X and kernel Y [8].

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].

$$(\boldsymbol{X} * \boldsymbol{K})(i,j) = \sum_m \sum_n X_{m,n} K_{i-m,j-m}$$

Since convolution is commutative, the following also holds.

$$(\boldsymbol{K} * \boldsymbol{X})(i,j) = \sum_m \sum_n X_{i-m,j-n} K_{m,n}$$

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].

$$(\boldsymbol{K} * \boldsymbol{X})(i,j) = \sum_m \sum_n X_{i+m,j+n} K_{m,n}$$

7.2 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].

7.3 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.

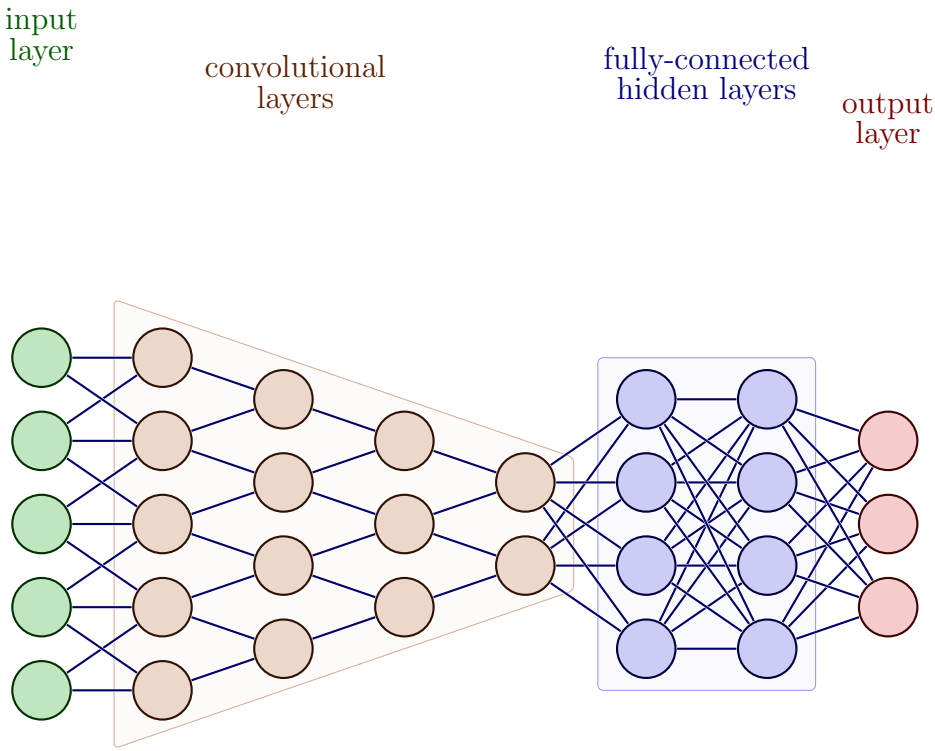


Figure 5: A deep convolutional network with four convolutional layers and two densely connected layers. [7].

7.4 Image Classification Example

In this example, we will use convolutional neural networks to perform image classification on the Rock Paper Scissors dataset. Other coding examples in this chapter use Pytorch and Lightning, but this example will use Tensorflow and Keras. While Pytorch is becoming the dominant Python framework for machine learning, Tensorflow is still a popular choice for deployment in industry. Using Tensorflow for this example also allows us to leverage the Tensorflow datasets library to fetch our data.

```
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
```

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

```
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.

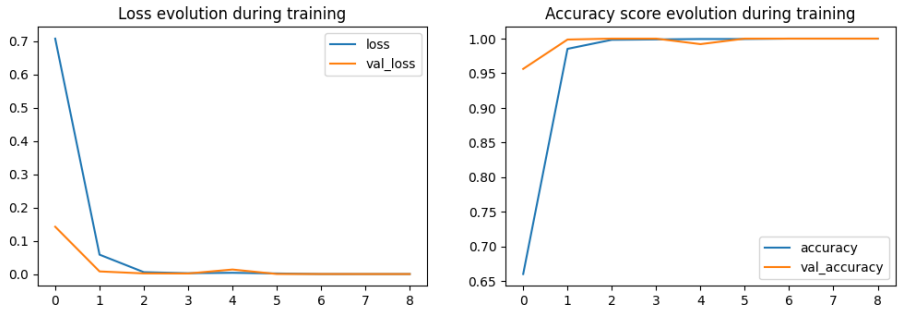
```
def plot_learning_curves(history):
    plt.figure(figsize=(12, 8))

    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")
```


Correct Predictions
True: 2, Pred: 2
True: 1, Pred: 1
True: 0, Pred: 0
True: 0, Pred: 0
True: 0, Pred: 0
True: 1, Pred: 1
True: 1, Pred: 1
True: 2, Pred: 2
True: 1, Pred: 1
True: 2, Pred: 2
True: 2, Pred: 2
True: 0, Pred: 0
True: 0, Pred: 0
True: 0, Pred: 0
True: 2, Pred: 2
True: 2, Pred: 2
True: 2, Pred: 2
True: 0, Pred: 0
True: 1, Pred: 1
True: 1, Pred: 1
True: 0, Pred: 0
True: 1, Pred: 1
True: 1, Pred: 1
True: 2, Pred: 2
True: 2, Pred: 2


Incorrect Predictions
True: 2, Pred: 1
True: 2, Pred: 1
True: 2, Pred: 1
True: 2, Pred: 1
True: 2, Pred: 1
True: 2, Pred: 1
True: 2, Pred: 1
True: 2, Pred: 1
True: 2, Pred: 1
True: 2, Pred: 1
True: 2, Pred: 1
True: 2, Pred: 1
True: 2, Pred: 1
True: 2, Pred: 1
True: 2, Pred: 1
True: 2, Pred: 1
True: 2, Pred: 1
True: 2, Pred: 1
True: 2, Pred: 1
True: 2, Pred: 1
True: 2, Pred: 1
True: 2, Pred: 1

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?

8 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.

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

RNNs are trained via backpropagation through time. Backpropagation through time involves "unrolling" the network across time steps and performing backpropagation as described in section 21.4.3.

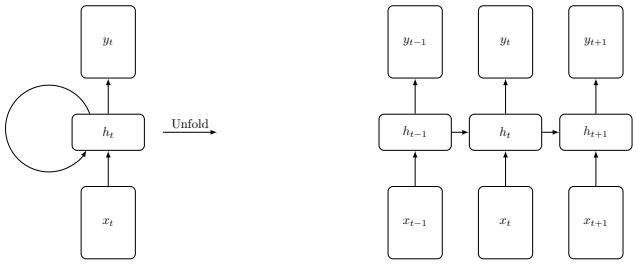


Figure 6: Your TikZ diagram inside a figure.

The four main types of recurrent neural networks are sequence-to-sequence, sequence-to-vector, vector-to-sequence, and encoder-decoder. We explore these networks in more detail below.

8.1 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 [6].

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

where the hidden state is

$$\begin{aligned} p(\boldsymbol{h}_t|\boldsymbol{h}_{t-1}, \boldsymbol{y}_{t-1}, \boldsymbol{x}) &= \mathbb{I}(\boldsymbol{h}_t = f(\boldsymbol{h}_{t-1}, \boldsymbol{y}_{t-1}, \boldsymbol{x})) \\ &= \begin{cases} 1 & \text{if } \boldsymbol{h}_t = f(\boldsymbol{h}_{t-1}, \boldsymbol{y}_{t-1}, \boldsymbol{x}) \\ 0 & \text{if } \boldsymbol{h}_t \neq f(\boldsymbol{h}_{t-1}, \boldsymbol{y}_{t-1}, \boldsymbol{x}) \end{cases} \end{aligned}$$

and the output function is $\boldsymbol{h}_t = \varphi(\boldsymbol{W}_{xh}[\boldsymbol{x}; \boldsymbol{y}_{t-1}] + \boldsymbol{W}_{hh}\boldsymbol{h}_{t-1} + \boldsymbol{b}_h)$ [6].

For categorical outputs, $p(\boldsymbol{y}_t|\boldsymbol{h}_t) = \text{Categorical}(\boldsymbol{y}_t|\text{softmax}(\boldsymbol{W}_{hy}\boldsymbol{h}_t + \boldsymbol{b}_y))$. For continuous outputs, $p(\boldsymbol{y}_t|\boldsymbol{h}_t) = N(\boldsymbol{y}_t|\boldsymbol{W}_{hy}\boldsymbol{h}_t + \boldsymbol{b}_y, \sigma^2\boldsymbol{I})$.

8.2 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].

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

8.3 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].

8.4 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.

8.5 Bidirectional RNNs

In some cases, we want an output at time t to depend on both future and past inputs. In this case, we use a bidirectional RNN. A birdirectional RNN consists of two RNNs: one "reads" inputs from right to left (computes hidden states for past inputs), and the other works from left to right (hidden states for future inputs) [6][2].

The forward and backward hidden states are computed

$$\boldsymbol{h}_t^{\rightarrow} = \varphi \boldsymbol{W}_{xh}^{\rightarrow} \boldsymbol{x}_t + \boldsymbol{W}^{r}ightarrow_{hh} \boldsymbol{h}_{t-1}^{\rightarrow} + \boldsymbol{b}_h^{\rightarrow}$$
$$\boldsymbol{h}_t^{\leftarrow} = \varphi \boldsymbol{W}_{xh}^{\leftarrow} \boldsymbol{x}_t + \boldsymbol{W}^{r}ightarrow_{hh} \boldsymbol{h}_{t+1}^{\leftarrow} + \boldsymbol{b}_h^{\leftarrow}$$

and $\boldsymbol{h}_t = [\boldsymbol{h}_t^{\rightarrow}, \boldsymbol{h}_t^{\leftarrow}]$

8.6 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].

8.6.1 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.

$$\begin{aligned}
\boldsymbol{I}_t &= \sigma(\boldsymbol{X}_t \boldsymbol{W}_{xi} + \boldsymbol{H}_{t-1} W_{hi} + \boldsymbol{b}_i) && \text{input gate} \\
\boldsymbol{F}_t &= \sigma(\boldsymbol{X}_t \boldsymbol{W}_{xt} + \boldsymbol{H}_{t-1} W_{hf} + \boldsymbol{b}_f) && \text{forget gate} \\
\boldsymbol{O}_t &= \sigma(\boldsymbol{X}_t \boldsymbol{W}_{xo} + \boldsymbol{H}_{t-1} W_{ho} + \boldsymbol{b}_o) && \text{output gate} \\
\tilde{\boldsymbol{C}}_t &= tanh(\boldsymbol{X}_t \boldsymbol{W}_{xc} + \boldsymbol{H}_{t-1} \boldsymbol{W}_{hc} + \boldsymbol{b}_c) && \text{candidate memory cell} \\
\boldsymbol{C}_t &= \boldsymbol{F}_t * \boldsymbol{C}_{t-1} + \boldsymbol{I}_t * \boldsymbol{tilde}\boldsymbol{C}_t && \text{final memory cell} \\
\boldsymbol{H}_t &= \boldsymbol{O}_t * tanh(\boldsymbol{C}_t) && = \text{final hidden state}
\end{aligned}$$

8.6.2 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].

$$\begin{aligned}
\boldsymbol{R_t} &= \sigma(\boldsymbol{X}_t\boldsymbol{W}_{xr} + \boldsymbol{H}_{t-1}\boldsymbol{W}_{hr} + \boldsymbol{b}_r) && \text{reset gate} \\
\boldsymbol{Z_t} &= \sigma(\boldsymbol{X}_t\boldsymbol{W}_{xz} + \boldsymbol{H_{t-1}}\boldsymbol{W}_{ht} + \boldsymbol{b}_z) && \text{update gate} \\
\tilde{\boldsymbol{H}}_{\boldsymbol{t}} &= tanh(\boldsymbol{H}_t - \boldsymbol{W}_{xh} + (\boldsymbol{R}_t * \boldsymbol{H_{t-1}})\boldsymbol{W}_{hh} + \boldsymbol{b}_h) && \text{candidate hidden state} \\
\boldsymbol{H}_t &= \boldsymbol{Z}_t * \boldsymbol{H}_{t-1} + (1 - \boldsymbol{Z}_t) * \boldsymbol{H}_t && \text{final hidden state}
\end{aligned}$$

8.7 Music Generation Example

In this example, we will use train an RNN with an LSTM layer on MIDI data, and then use our trained network to generate new music.

First, import dependencies and set seed to ensure reproductibility.

```
import os
import glob
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import pretty_midi
from torch.utils.data import Dataset, DataLoader,
    random_split
from lightning.pytorch.callbacks.early_stopping import
    EarlyStopping
import lightning as L
import torch.nn.functional as F
```

The MIDI data will be represented as a sequence of notes, each with three different features: pitch, step, and duration. Pitch is a categorical variable with 128 classes, each representing a different pitch that can be stored in MIDI format. Step is a continuous variable, repreesenting the time in beats between the note in question and the previous note. In accordance with musical theory best practices quantize this to a precision of 0.125. Duration is continous and represents how long the note is held. It is quantized in the same way as step.

Data-handling for this example is more complicated than for the previous examples. Data managment classes have been provided in the interest of providing complete instructions, but feel free to skip the code for this section if you are more interested in the implementation of them network itself.

The `MidiDataset` class loads all of the MIDI files in a given directory into a dataset. For each file, overlapping instrument tracks are combined into one

acoustic grand piano track. Pitches and durations are quantized, and the sequences are concatenated.

```
class MidiDataset(Dataset):
    def __init__(self, midi_dir: str, seq_length: int,
    quantize_step=0.125, quantize_duration=0.125, vocab_size
    =128):
        self.midi_dir = midi_dir
        self.seq_length = seq_length
        self.quantize_step = quantize_step
        self.quantize_duration = quantize_duration
        self.vocab_size = vocab_size
        self.data = self.load_and_process_data()

    def load_and_process_data(self):
        midi_files = glob.glob(os.path.join(self.midi_dir, '
    *.mid'))
        data = []
        for midi_file in midi_files:
            merged_midi_data = self.merge_tracks(midi_file)
            notes = sorted(merged_midi_data.instruments[0].
    notes, key=lambda note: note.start)
            processed_notes = self.process_notes(notes)
            for i in range(len(processed_notes) - self.
    seq_length):
                data.append(processed_notes[i:i + self.
    seq_length + 1])
        return data

    def merge_tracks(self, file_path):
        midi_data = pretty_midi.PrettyMIDI(file_path)
        merged_midi = pretty_midi.PrettyMIDI()
        merged_instrument = pretty_midi.Instrument(program
    =0)

        all_notes = []
        for instrument in midi_data.instruments:
            all_notes.extend(instrument.notes)

        sorted_notes = sorted(all_notes, key=lambda note:
    note.start)
        merged_instrument.notes.extend(sorted_notes)
        merged_midi.instruments.append(merged_instrument)

        return merged_midi

    def process_notes(self, notes):
        processed = []
        for i in range(1, len(notes)):
            pitch = notes[i].pitch
```

```
            step = round((notes[i].start - notes[i-1].start)
    / self.quantize_step) * self.quantize_step
            duration = round((notes[i].end - notes[i].start)
    / self.quantize_duration) * self.quantize_duration
            processed.append([pitch, step, duration])
        return processed

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        sequence = self.data[idx]
        inputs = torch.tensor(sequence[:-1], dtype=torch.
    float32)
        pitch_target = torch.tensor(sequence[-1][0], dtype=
    torch.long)
        step_target = torch.tensor(sequence[-1][1], dtype=
    torch.float32)
        duration_target = torch.tensor(sequence[-1][2],
    dtype=torch.float32)
        return inputs, pitch_target, step_target,
    duration_target
```

The `MidiDataModule` class allows us to use the dataset with Lightning, and handles the train and validation splits.

Why aren't we using a test set? Our goal is not to predict a "correct" outcome; it is to generate outputs that sound nice (or at the very least interesting). A test set is therefore not necessary.

```
class MidiDataModule(L.LightningDataModule):
    def __init__(self, midi_dir: str, seq_length: int,
    batch_size: int, val_split=0.2, quantize_step=0.125,
    quantize_duration=0.125):
        super().__init__()
        self.midi_dir = midi_dir
        self.seq_length = seq_length
        self.batch_size = batch_size
        self.val_split = val_split
        self.quantize_step = quantize_step
        self.quantize_duration = quantize_duration

    def setup(self, stage=None):
        full_dataset = MidiDataset(self.midi_dir, self.
    seq_length, self.quantize_step, self.quantize_duration)

        val_size = int(len(full_dataset) * self.val_split)
        train_size = len(full_dataset) - val_size
        self.train_dataset, self.val_dataset = random_split(
    full_dataset, [train_size, val_size])
```

```
    def train_dataloader(self):
        return DataLoader(self.train_dataset, batch_size=
    self.batch_size, shuffle=True)

    def val_dataloader(self):
```

Now we can define our model class. Pitch is categorical, so we use cross entropy for the pitch loss. Step and duration are continuous, so for those we use MSE. Our first layer is an LSTM layer, followed by a dropout layer for regularization. Each feature has a corresponding output layer. We also implement weight decay with our optimizer (Adam).

```
class MIDIModel(L.LightningModule):
    def __init__(self, input_size, hidden_size, output_size,
     dropout_rate=0.1, weight_decay=1e-5):
        super(MIDIModel, self).__init__()
        self.rnn = nn.LSTM(input_size, hidden_size,
    batch_first=True) # LSTM layer

        self.dropout = nn.Dropout(dropout_rate) # dropout
    layer

        self.fc_pitch = nn.Linear(hidden_size, output_size)
    # pitch output
        self.fc_step = nn.Linear(hidden_size, 1) # step
    output
        self.fc_duration = nn.Linear(hidden_size, 1) #
    duration output

        self.loss_fn_pitch = nn.CrossEntropyLoss() # pitch
    loss
        self.loss_fn_step_duration = nn.MSELoss() # step and
     duration loss

        self.weight_decay = weight_decay

    def forward(self, x):
        rnn_out, _ = self.rnn(x)
        rnn_out = self.dropout(rnn_out)
        pitch_out = self.fc_pitch(rnn_out[:, -1, :])
        step_out = self.fc_step(rnn_out[:, -1, :])
        duration_out = self.fc_duration(rnn_out[:, -1, :])

        return {'pitch': pitch_out, 'step': step_out, '
    duration': duration_out}

    def training_step(self, batch, batch_idx):
        inputs, pitch_target, step_target, duration_target =
     batch
```

```
        predictions = self(inputs)

        loss_pitch = self.loss_fn_pitch(predictions['pitch'
    ], pitch_target)
        loss_step = self.loss_fn_step_duration(predictions['
    step'], step_target.unsqueeze(-1))
        loss_duration = self.loss_fn_step_duration(
    predictions['duration'], duration_target.unsqueeze(-1))

        loss = loss_pitch + loss_step + loss_duration
        self.log('train_loss', loss, prog_bar=True, logger=
    True) # log the training loss

        return loss

    def validation_step(self, batch, batch_idx):
        inputs, pitch_target, step_target, duration_target =
     batch
        predictions = self(inputs)

        loss_pitch = self.loss_fn_pitch(predictions['pitch'
    ], pitch_target)
        loss_step = self.loss_fn_step_duration(predictions['
    step'], step_target.unsqueeze(-1))
        loss_duration = self.loss_fn_step_duration(
    predictions['duration'], duration_target.unsqueeze(-1))

        loss = loss_pitch + loss_step + loss_duration
        self.log('val_loss', loss, prog_bar=True, logger=
    True) # log the validation loss

        return loss

    def configure_optimizers(self):
        optimizer = optim.Adam(self.parameters(), lr=0.001,
    weight_decay=self.weight_decay)

        return optimizer
```

Next, we define necessary variables and initialize our data module.

```
midi_dir = "/my-midi-directory" # replace with the path to
    the folder containing your MIDI files
seq_length = 25
batch_size = 64
vocab_size = 128
input_size = 3
hidden_size = 128
output_size = vocab_size

data_module = MidiDataModule(midi_dir=midi_dir, seq_length=
```

```
    seq_length, batch_size=batch_size)
```

Now we can initialize and train our model. In the interest of preserving cloud computation resources we only train for 20 epochs, but the early stop callback would faciliate longer training if desired.

```
early_stop_callback = EarlyStopping(monitor="val_loss",
    min_delta=0.00, patience=3, verbose=False, mode="min")
trainer = L.Trainer(max_epochs=20, callbacks=[
    early_stop_callback], accelerator="gpu" if torch.cuda.
    is_available() else "cpu", devices=1)

trainer.fit(model, data_module)
```

```
INFO: LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
INFO: lightning.pytorch.accelerators.cuda: LOCAL_RANK: 0 -
    CUDA_VISIBLE_DEVICES: [0]
INFO:
| Name                   | Type             | Params | Mode
-------------------------------------------------------------------

0 | rnn                    | LSTM             | 68.1 K |
    train
1 | dropout                | Dropout          | 0      |
    train
2 | fc_pitch               | Linear           | 16.5 K |
    train
3 | fc_step                | Linear           | 129    |
    train
4 | fc_duration            | Linear           | 129    |
    train
5 | loss_fn_pitch          | CrossEntropyLoss | 0      |
    train
6 | loss_fn_step_duration | MSELoss          | 0      |
    train
-------------------------------------------------------------------

84.9 K    Trainable params
0         Non-trainable params
84.9 K    Total params
0.339     Total estimated model params size (MB)
7         Modules in train mode
0         Modules in eval mode
INFO: lightning.pytorch.callbacks.model_summary:
| Name                   | Type             | Params | Mode
-------------------------------------------------------------------

0 | rnn                    | LSTM             | 68.1 K |
    train
1 | dropout                | Dropout          | 0      |
    train
```

```
2 | fc_pitch              | Linear           | 16.5 K |
    train
3 | fc_step               | Linear           | 129    |
    train
4 | fc_duration           | Linear           | 129    |
    train
5 | loss_fn_pitch         | CrossEntropyLoss | 0      |
    train
6 | loss_fn_step_duration | MSELoss          | 0      |
    train
----------------------------------------------------------------------

84.9 K    Trainable params
0         Non-trainable params
84.9 K    Total params
0.339     Total estimated model params size (MB)
7         Modules in train mode
0         Modules in eval mode
Epoch 19: 100% [============================] 1885/1885
    [00:17<00:00, 109.42it/s, v_num=0, train_loss=3.410,
    val_loss=3.590]
```

Now that our model is trained, we can use it to generate new music. The `get_seed-sequence` function allows us to sample a seed sequence from a different MIDI file to be used as input. Alternatively, a custom sequence could be defined manually.

```
def get_seed_sequence(midi_file_path, device, num_notes=25,
    skip_notes=25, quantize_step=0.125, quantize_duration
    =0.125):
    midi_data = pretty_midi.PrettyMIDI(midi_file_path)

    merged_notes = []
    for instrument in midi_data.instruments:
        merged_notes.extend(instrument.notes)

    notes = sorted(merged_notes, key=lambda note: note.start
    )

    if len(notes) < (skip_notes + num_notes):
        print(f"Not enough notes after skipping the first {
    skip_notes}. Using all available notes after skipping.")
        start_index = skip_notes if skip_notes < len(notes)
    else 0
        num_notes = len(notes) - start_index
    else:
        start_index = skip_notes

    seed_sequence = []
    previous_start = notes[start_index].start if notes else
    0.0
```

```
    for note in notes[start_index:start_index + num_notes]:
        pitch = note.pitch

        step = note.start - previous_start
        duration = note.end - note.start

        step = round(step / quantize_step) * quantize_step
        duration = round(duration / quantize_duration) *
    quantize_duration

        seed_sequence.append([pitch, step, duration])
        previous_start = note.start

    seed_sequence_tensor = torch.tensor(seed_sequence, dtype
    =torch.float32).to(device)
    return seed_sequence_tensor

midi_file_path = "path-to/seed-file.mid" # make sure this
    isn't part of the training / validation dataset
seed_sequence = get_seed_sequence(midi_file_path, device=
    device, num_notes=25, quantize_step=0.125,
    quantize_duration=0.125)
```

Now we can generate music! Pitch generation is probabilistic, with randomness controlled by the temperature parameter (higher temperature leads to more randomness). Step and duration generation are deterministic.

```
def generate_notes(model, device, seed_sequence,
    sequence_length, vocab_size=128, temperature=1.0,
    quantize_step=0.125, quantize_duration=0.125, max_silence
    =1.0):
    model.eval()  # set model to evaluation mode
    generated_sequence = []

    current_sequence = seed_sequence  # Initialize with the
    seed sequence

    for _ in range(sequence_length):
        current_sequence_tensor = current_sequence.unsqueeze
    (0).to(device)

        with torch.no_grad():
            predictions = model(current_sequence_tensor)

        pitch_logits = predictions['pitch'].squeeze() /
    temperature # apply temperature scaling

        pitch_probs = F.softmax(pitch_logits, dim=-1)

        predicted_pitch = torch.multinomial(pitch_probs,
```

```
    num_samples=1).item() # sample pitch from multinomial
    distribution

        predicted_step = predictions['step'].squeeze().item
    ()
        predicted_duration = predictions['duration'].squeeze
    ().item()

        predicted_step = min(predicted_step, max_silence) #
    max silence

        predicted_step = round(predicted_step /
    quantize_step) * quantize_step

        predicted_duration = round(predicted_duration /
    quantize_duration) * quantize_duration

        generated_sequence.append([predicted_pitch,
    predicted_step, predicted_duration])

        new_note = torch.tensor([[predicted_pitch,
    predicted_step, predicted_duration]], dtype=torch.float32
    ).to(device)
        current_sequence = torch.cat((current_sequence[1:],
    new_note))

    return generated_sequence

model.to(device)
generated_sequence = generate_notes(model, device=device,
    seed_sequence=seed_sequence, sequence_length=400,
    temperature=1)
```

Finally, the big reveal... we output our generated sequence to a MIDI file, which can then be played back.

```
def notes_to_midi(generated_sequence, output_path="
    generated_music.mid"):
    midi = pretty_midi.PrettyMIDI()
    instrument = pretty_midi.Instrument(program=0)

    start_time = 0

    for note in generated_sequence:
        pitch = int(note[0])
        step = note[1]
        duration = note[2]

        start_time += step
        end_time = start_time + duration
```

```
        midi_note = pretty_midi.Note(
            velocity=100,
            pitch=pitch,
            start=start_time,
            end=end_time
        )
        instrument.notes.append(midi_note)

    midi.instruments.append(instrument)
    midi.write(output_path)

notes_to_midi(generated_sequence, output_path="
    generated_music.mid")
```

Find the output file in your file explorer and listen to your generated sequence. Does it sound like music? Is it good music? How do you think the model could be improved? Try experimenting, different regularization techniques, additional layers and investigating the parameter losses individually.

9 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.

9.1 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

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

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, Bengio, and Courville and chapter 26 of *Probabilistic Machine Learning: Advanced Topics* by Murphy.

9.2 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, Bengio, and Courville 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] Izaak Neutelings. *Neural networks*. Licensed under the Creative Commons Attribution 4.0 International License. To view a copy of this license, visithttp://creativecommons.org/licenses/by/4.0/. Changes were made to this work. 2022. URL: https://tikz.net/neural_networks/#full_code.

[8] Janosh Riebesell and Stefan Bringuier. *Collection of standalone TikZ images*. Version 0.1.0. MIT License Copyright (c) 2021 Janosh Riebesell Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions: The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software. The software is provided "as is", without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose and noninfringement. In no event shall the authors or copyright holders be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the software or the use or other dealings in the software. Aug. 9, 2020. DOI: 10.5281/zenodo.7486911. URL: https://github.com/janosh/tikz.

[9] 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.

[10] 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.

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