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 Architecture Design

3.1 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 [3].

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.

3.2 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 [2]. If all activation functions within a deep network are linear, then the network reduces to a linear model [4]. 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 [2].

3.2.1 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" [4].

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 [2]. 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 [2].

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 [1].

Parametric Leaky ReLU Leaky ReLU where *alpha* is learned by the model [1].

3.2.2 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.

4 Training Deep Neural Networks

4.1 Hyperparameter Tuning

4.2 Regularization

4.2.1 L^2 Regularization

4.2.2 L^1 Regularization

4.2.3 Batch Normalization

4.2.4 Layer Normalization

4.2.5 Early Stop

4.2.6 Weight Decay

4.2.7 Dropout

4.2.8 Unsupervised Pretraining

4.2.9 Semi-Supervised Learning

4.3 Optimization

5 Convolutional Neural Networks

5.1 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

$$\underbrace{s(i)}_{\substack{\text{feature}\\ \text{map}}} = (x * y)(i) = \sum_m \underbrace{x(m)}_{\substack{\text{input}\\ \text{data}}} \underbrace{y(i-m)}_{\text{kernel}} \tag{11}$$

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 [2].

$$s(i,j) = (x * y)(i,j) \sum_m \sum_n x(i,j) y(i-m, j-n) \tag{12}$$

Since convolution is commutative, equation 13 also holds.

$$s(i,j) = (y * x)(i,j) \sum_m \sum_n x(i-m, j-n) y(i,j) \tag{13}$$

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 [2].

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

5.2 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 [2].

5.3 The Convolutional Layer

The convolutional layers used in a CNN have three steps [2].

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.

5.4 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]
```

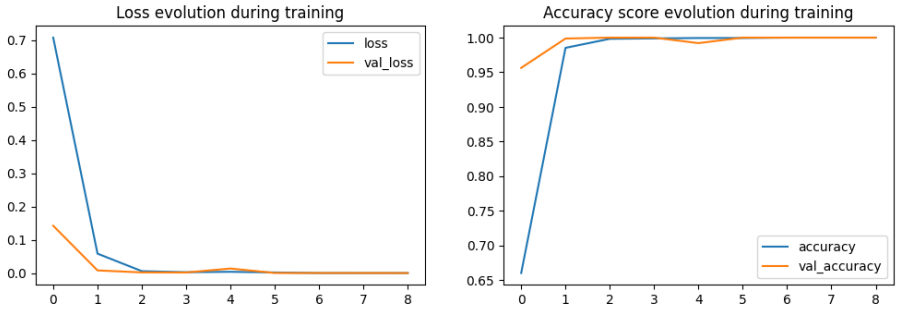
Now that we have reached the accuracy threshold, let's plot the learning curves.

```
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?

6 Recurrent Neural Networks

7 Generative Models

7.1 Generative Adversarial Neural Networks

7.2 Variational Autoencoders

7.3 Quantization