Abschlussbericht zur Projektarbeit II:

Optimierung von Convolutional Neural Network Modellen zur Prediction von Wasserversorgung, Nahrungsmittelversorgung oder Wohlstandsindizes

Von

Vera Bateva und Julia Przibylla

Prüfer: **Prof. Dr. Gernot Heisenberg**

Sven Wöhrle, MSc.

Datum: 15.07.2022

Inhalte

1. **Freezing / Unfreezing**
2. **Fine-Tuning**
3. **Test Routines**
4. **Lessons Learned**

Links:

Github: dis22_ss22_bateva_przibylla

1. Freezing / Unfreezing

Scripts:

- *congif.py*
- *water_main.py*
- *nn_models.py*

Aufgabe:

Analyse / Auswertung der Auswirkung der Anzahl bzw. des Anteils des Parameters "*ungefroreren Layer*" auf die Performance des vortrainierten Models.

Das Einfrieren soll vermeiden, dass die in ihnen enthaltenen Informationen bei künftigen Trainingsrunden zerstört werden.[1] Zudem reduziert es die Trainingszeit, da die Gewichte der gefrorenen Schichten nicht geupdated werden müssen.

Zudem lag für uns eine Herausforderung dabei, anhand "unfreeze_precentage" die Anzahl der Layer zu bestimmen. Die Prozenangabe entspricht der Anzahl der Layer und nicht der Parameter, in Tabelle 1 sehen wir, dass wenn wir "unfreeze_precentage" um 4 Prozentpunkte erhöhen, sich der Wert der trainierbaren Parameter des Modells nur um Prozentpunkte im Bereich von 0,01 bis 0,1 erhöht. Daher haben wir uns zusätzliche die Aufgabe gestellt weitere Konfigurationen zu implementieren, d.h. die genaue Anzahl der Schichten oder die genauen Blöcke auswählen zu können.

Strategie:

Da es sich empfiehlt nur die Top-Layer "aufzutauen" haben wir zunächst anhand des Parameters "unfreeze_precentage" identifiziert, bis zu welchem Layer wir einfrieren möchten. Die obersten 5 Layer liegen in den obersten 16%.

Um die zusätzlichen Konfigurationen zu implementieren haben wir ein Dictionary ausgewählt, welches diese enthält. Dieses kann leichter im Script der *water_main.py* eingefügt werden, da es mehrere Variablen enthält.

Umsetzung:

Eingefügt in *config.py* ,

```
unfreeze_dict = {"unfreeze_type": "unfreeze_blocks",
        "unfreeze_layers_perc": 86, # Use custom top layers (necessary when using transferlearning)
        "unfreeze_at": 17,
        "unfreeze_blocks": ['input', 'block1']}
```

In *nn_models.py*

```
  if unfreeze_dict:
     if unfreeze_dict['unfreeze_type'] == 'unfreeze_layers_perc':
        # freeze layers of input model
        # unfreeze at least one layer if unfreeze layers != 0
        unfreeze_layers = unfreeze_dict['unfreeze_layers_perc']
        #freeze layers of input model
        #unfreeze at least one layer if unfreeze layers != 0
        unfrozen_layers = max(1, round(len(model.layers) * unfreeze_layers/100))
        freeze_layers = len(model.layers) - unfrozen_layers
        for layer in model.layers[0:freeze_layers]:
           layer.trainable = False
        print('Frozen layers', freeze_layers, 'unfrozen layers', unfrozen_layers, 'ges_layers',
len(model.layers))
     elif unfreeze_dict['unfreeze_type'] == 'unfreeze_at':
        # Instead of freezing at a percentage of layers, select the number of layers
        if len(model.layers) > unfreeze_dict['unfreeze_at']:
           unfreeze_at = unfreeze_dict['unfreeze_at']
           print("Number of the maximum layer exceeded")
        # If 'unfreeze_at' exceeds the maximum value take the maximum value
        else:
           unfreeze_at = len(model.layers)
        freeze_layers = len(model.layers) - unfreeze_at
        for layer in model.layers[0:unfreeze_at]:
           layer.trainable = False
        print('Frozen layers', freeze_layers, 'unfrozen at', unfreeze_at, 'ges_layers',
len(model.layers))
```

```
        elif unfreeze_dict['unfreeze_type'] == 'unfreeze_blocks':
            # Instead of freezing from the first one, you can choose blocks from anywhere
            unfrozen_blocks = unfreeze_dict['unfreeze_blocks']
            for layer in model.layers:
                layer_name = str(layer.name)
                layer_name = layer_name.split("_")[0]
                if layer_name in unfrozen_blocks:
                    layer.trainable = False
                else:
                    layer.trainable = True
            print('Unfrozen block(s)', unfrozen_blocks, 'ges_layers', len(model.layers))
        # (https://medium.com/@timsennett/unfreezing-the-layers-you-want-to-fine-tune-using-transfer-learning-1bad8cb72e5d)
```

und eingesetzt in *model.py*

```
if cfg.cnn_settings_d['include_top'] is False and cfg.add_custom_top_layers:
        base_model = nnm.add_classification_top_layer(base_model, cfg.cnn_settings_d['classes'], cfg.neurons_l, cfg.unfreeze_dict, cfg.dropout_top_layers)
```

Ergebnisse:

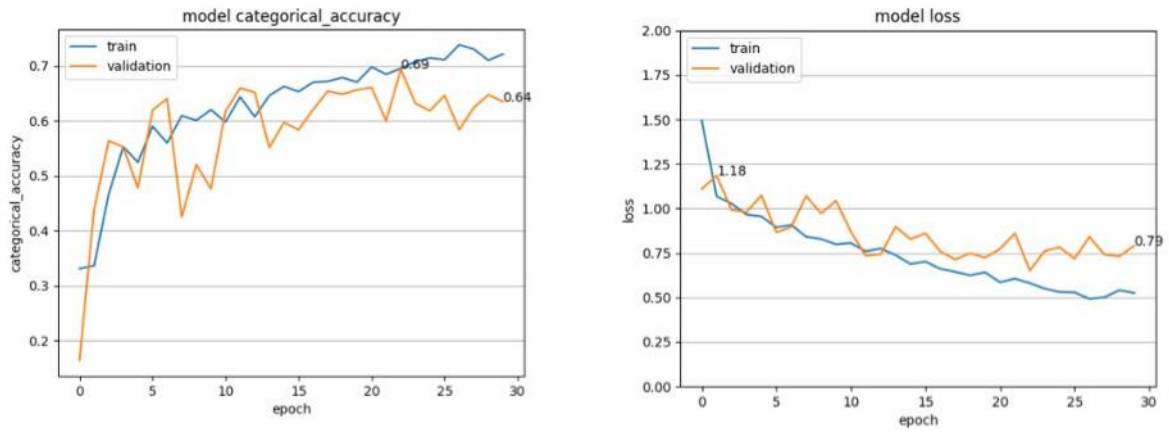
In der folgenden Tabelle sieht man die Ergebnisse aus unserem ersten Durchlauf.

Tabelle 1: "Ergebnisse der Testroutinen für unfrozen_percantage."

Unfreezing %	Unfrozen Layer	IAug. (X)	Epochs (Early Stopping*)	Trainable Param (M)	Trainable Param %	(Test) Acc	(Test) Loss	Max Epochs
84	22		30 (29)	21,037	99,9%	0.28	1.13	9
88	23		30	21,037	99,9%	0.66	0.93	13
92	24		30 (21)	21,074	100%	0.66	1.01	1
96	**25**		**30 (21)**	**21,076**	**100%**	**0.7**	**0.94**	**1**
100 (fine-tuning)	26		30 (21)	21,076	100%	0.66	1.07	1

Wir haben den Wert für Early Stopping auf 20 heruntergesetzte, deshalb haben die Modelle bereits so früh mit dem Training ausgesetzt. Dennoch sieht man in der Abbildung 1, die Modelle könnten somit noch nicht austrainiert sein. Andererseits sieht man ca. Ab Epoche 22 eine Tendenz zum Overfitting. Das beste Ergebnis erhalten wir bei 96% unfrozen Layers.

Abbildung 1: “unfreeze_percantage = 96”



Problem:

Da die Performance mehrerer vortrainierten Modelle, die unterschiedlich groß sind (unterschiedlich viel Layers haben), untersucht wird, wurden die ungefroreren Layers als Prozentzahlen angegeben. Dadurch war zunächst nicht klar, welche Layers bzw. Blocks gefroren blieben und welche trainiert würden oder wieviel Layers das eigentlich wären. Es war möglich sich an der Anzahl der trainerbaren Parameter zu orientieren, die aber bei kleineren Intervallen doch gleichblieben. Um dieses Problem bzw. diese Unklarheit umzugehen, wurde ein Dictionary mit den verschiedenen Arten zu Angabe der Layers, die ungefroren werden sein müssen und nämlich:

2. Fine-Tuning

Scripts:

- *congif.py*
- *water_main.py*

Ziel:

Implementierung des Fine-Tuning Ansatzes aus [2]:

> *“In fine tuning, the first step (part of the model is frozen) is trained for 50 epochs, and the second step (all layers of the model are free to learn) for another 50 epochs.*
>
> *[…]*
>
> *In fine-tuning achieved results are comparable to **or better** than results from training a CNN model with randomly initialized weights.”*

Strategie:

Das verketten zweier Modell-Durchläufe mit verschiedenen Parametern (Freezing), sowie die der *history* Objekte.

Umsetzung:

Eingesetzt in *config.py*

```
################################################################################
#                                Fine-Tuning
################################################################################
fine_tuning = True # If you set fine-tuning as true you will run 2 iterations of the model
epochs_freezing = 2 # Number of epochs you want to train with the aforementioned settings
epochs_unfreezing = 4 # Number of Epochs you want to train unfreezed.
```

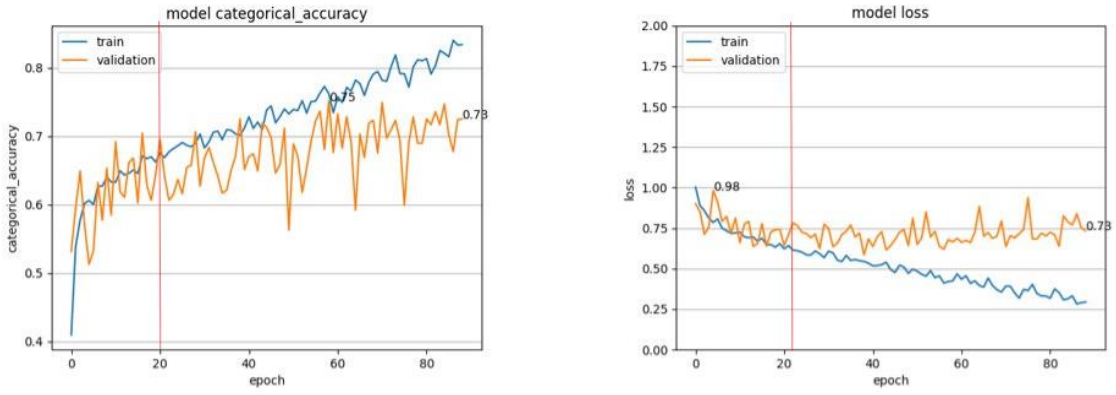
Eingesetzt in *water_main.py*

```
if cfg.fine_tuning:
    # create empty lists to append values from each iterationn
    acc, val_acc, loss, val_loss = []
    for i in (0,1):
        if i == 0:
            with strategy.scope():
                # Everything that creates variables should be under the strategy scope.
                # In general this is only model construction & `compile()`.
                # Load Model
                model = model_loader()
                # Create Metrics
                metrics_l = [tf.keras.metrics.CategoricalAccuracy()]
                model.compile(optimizer=optimizer, loss=cfg.loss, metrics=[metrics_l])
                cfg.epochs = cfg.epochs_freezing
        if i == 1:
            i = i - 1
            model.trainable = True #set trainable true for the second iteration
            cfg.epochs = cfg.epochs_unfreezing
        if cfg.verbose:
            print('Final Model', type(model))
            print(model.summary(show_trainable = True))
            # check the model settings for trainable


        ###Run Model
        t_begin = time.time()


        history = model.fit( [...])
```

```
    # extend the list with all new model metrics
    acc.extend(history.history['categorical_accuracy'])
    val_acc.extend(history.history['val_categorical_accuracy'])
    loss.extend(history.history['loss'])
    val_loss.extend(history.history['val_loss'])
t_begin = time.time()


history = model.fit( [...])


fit_time = time.time() - t_begin
if cfg.verbose:


    print('Time to fit model (s)', fit_time)


history.history['categorical_accuracy'] = acc
history.history['val_categorical_accuracy'] = val_acc
history.history['loss'] = loss
history.history['val_loss'] = val_loss
```

Ergebnisse:

<u>Abbildung 2:</u> “Fine-Tuning: 20 Epochen erster Block unfrozen und anschließend 100% trainable.”



Bei den Ergebnissen wird deutlich, dass die Modelle Überanpassen, sobald diese in den freien Trainingsmodus übergehen. Wir gehen aber davon aus, dass wir nicht die optimalen Hyperparamater verwendet haben, da wir das Grid-Testing erst parallel ausgeführt haben.

Vergleicht man die Ergebnisse, mit denen aus Abbildung 1. sehen wir dennoch eine Verbesserung der Performance und eine Verminderung des Loss.

Probleme und Ausblick:

Zunächst die Begriffsdifferenzierung von *"Fine-Tuning"* und *"Ensemble Learning"*. Wir haben uns zunächst mit Ensemble Learning Methoden befasst, und uns dann wieder explizit auf die Literatur bezogen, in der ein Fine-Tuning Ansatz verfolgt wurde. Beide Methoden verfolgen das Ziel entweder verschiedene Modelle oder die Ergebnisse verschiedener Methode zu aggregieren, um einen sogenannten "schwachen Lerner" zu verbessern. Zukünftig wäre es somit interessant das Thema "Ensemble Learning" aufzugreifen. Mehr dazu haben wir in https://teams.microsoft.com/ dokumentiert.

3. Test Routines

Scripts:

- *congif.py*
- *water_main.py*

Ziel:

Script schreiben, der alle Testroutinen vereinigt. Dabei sollten *„lr"*, *„dropout_top_layers"*, *„unfreeze_layers_perc"* sowie *„IDG_augmentation_settings_d"* betrachtet werden. Das Ziel ist einen Script zu haben, der bei einem aktiven Test-Mode ausgeführt wird und das Model automatisiert mit verschiedenen Parameter-Kombinationen trainiert werden kann. Zudem sollte die Möglichkeit eingebaut werden, dass nur einzelne Hyperparameter auf Test-Mode gesetzt werden können.

Strategie:

Zusammenfügen aller Hyperparameter, mit denen das Model, im Rahmen eines Testings (`testing = True`), trainiert werden soll. Im nächsten Schritt werden daraus verschiedene Kombinationen/Sets mit jeweils *„lr", „dropout_top_layers", „unfreeze_layers_perc"* und *„IDG_augmentation_settings_d"* gebildet und in einer Liste gespeichert. Wird der Testmode auf aktiv gesetzt (`testing = True`), wird der *water_main.py* mit jeweils einem Element dieser Liste (= Set aus Hyperparametern) ausgeführt.

Umsetzung:

Es wurden zwei Sections im *config.py* hinzugefügt – Testsettings und Testcode.

- Testsettings:

  Hier wird eingestellt, ob ein Testlauf gestartet werden muss. Dafür wurde die Variable `testing` definiert, die die Werte `True` und `False` einnehmen kann. Ist der Testmode aktiviert, wird der *water_main.py* mit den in Section Tescode generierten Hyperparametern ausgeführt.

  Als nächstes kann man über mode bestimmen, in welchem in welchem Modus bzw. mit welchem Hyperparameterset das Model trainiert werden muss. Aktuell ist nur der Modus „all" lauffähig. (Siehe *Probleme und Ausblick*)

  Über einzelne Variablen (test_lr, test_unfreeze_layers_perc, test_dropout_top_layers, test_IDG_augmentation_settings_d) hat man die Möglichkeit, einzelne Parameter in Test-Mode zu setzen. Dies erfolgt wieder mit `True` und `False`.

- Testcode

Die Hyperparameter, die für die Testzwecken von Interesse sind, werden einem Dictionary übergeben. Die Keys entsprechen den Hyperparameter und die Values – ihre Werte. Aus diesen Values werden anschließend alle möglichen Kombinationen erstellt.

Für die Augmentation Settings wird ein separates verschachteltes Dictionary definiert, mit Ausblick auf die Zukunft, wenn neue Subsets dazu kommen sollten.

Die Hypermarameter-Kombinationen werden als letztes mit jeweils einem Subset aus dem Augmentation-Settings-Dictionary vereinigt und in einer Liste gespeichert. Wenn der *water_main.py* in Test-Mode läuft, wird dieser durch die Liste iterieren und einen Durchlauf mit jedem Set ausführen.

Ergebnisse:

Im Testmodus läuft das Script fehlerfrei. Wir haben einen Testlauf, mit dem wir das Model mit 76 verschiedenen Parameter-Kombinationen trainieren und evaluieren konnten. Das Modell wurde für 20 Epochen trainiert. Folgende Parameter wurden verwendet:

```
model_test_param = {"learning_rates": [0.001, 0.01, 0.1],
                    "dropout_top_layers": [0.3, 0.4, 0.5],
                    "unfreezed_layers_perc": [20, 40, 60, 80]}
```

Kombiniert mit den beiden im config.py angegebenen Augmentation-Settings-Subsets.*

* Der zweite Subset wurde eingefügt, damit wir testen, ob der Script fehlerfrei läuft. Die Werte sind nicht unbedingt plausibel.

Der beste run hatte einen Test Accuracy i.H.v. 0.7531 mit einem Test Loss von 0.5669 erreicht. Dabei wurden einen *„learning rate“* von 0.001 und einen *„dropout_top_layer“* von 0.4 verwendet und des wurden 40% der Layers ungefroren.

Trotz der vielen Durchläufe war in den Ergebnissen kein Muster zu erkennen, auf dessen Basis man einen Rückschluss ziehen könnte. Um sagen zu können, wie die Hyperparameter zusammenhängen und welche Kombinationen die beste Performance erreichen, müssen weitere Testläufe durchgeführt werden. Aus zeitlichen Gründen konnten jedoch nicht alle mögliche Kombinationen getestet werden. Der einzige Hyperparameter, der die ganze Zeit eine führende Position annimmt, ist der *„learning rate“* von 0.001.

Probleme und Ausblick:

Zum Teil liegen uns nur die Parameter („learning rate“, „dropout_top_layer“, „IDG_augmentation_settings_d“) vor, mit denen die einzelnen Gruppen ihren besten Run erzielt hatten. Für die Vereinigung aller Testroutinen sowie zum Testzweck (zum einen Evaluieren der Performance des Models, zum anderen Evaluieren der Performance unseres Testing-Scripts) brauchten wir allerdings mehrere Hyperparameterwerte, die wir als Values unserem Dictionary übergeben. Aus diesem Grund haben wir den Dictionary teilweise um ”unsere” Werte erweitert. Außerdem haben alle Gruppen bei der ersten Aufgabe (Testroutinen mit Variieren der einzelnen Hyperparameter) das Modell für unterschiedliche Epochen trainiert, deshalb bleibt es fragwürdig, inwiefern die erzielten Ergebnisse untereinander vergleichbar sind und ob die besten Runs auch kombiniert die beste Performance des Models liefern könnten. Daher wäre es besser gewesen, wenn man von Anfang an, festgelegt hätte, für wieviel Epochen das Model trainiert werden soll und welche Werte die Parameter (die bei den Testroutinen nicht geändert werden) einnehmen sollen.

Was wir noch vor hatten, aber nicht umsetzten konnten, war ein Testing im Random-Mode zu erstellen, der der weiteren Optimierung des Modells dienen sollte.[3] Mithilfe vom dem Test-Mode „all“ kann es bestimmt werden, mit welchen Hyperparametern gute Ergebnisse erreicht werden. Wenn die Range der einzelnen dafür ”zuständigen” Hyperparameter bekannt ist, könnte die Liste der Parameter verkleinert und verfeinert werden. Hier wird der Random-Mode eingesetzt, der zufällige Werte aus den neudefinierten Bereichen auswählt und dem Model zum Trainieren übergibt.

Lessons Learned

Mad

- **Am Anfang haben wir viel Zeit verloren, da wir nicht mit dem eigentlichen Projektcode arbeiten konnten.**
- **Zu dem Code gab es leider keine Einführung, sodass wir zunächst nur langsame Fortschritte gemacht haben.**

Sad

- **Für das Aufsetzen der Testroutinen haben wir zwar die besten Ergebnisse der Gruppen gesammelt, allerdings waren diese nicht Vergleichbar, da die Gruppe z.B. unterschiedliche Epochen gewählt haben. Beim nächsten Mal würden wir darauf achten, dass wir unsere Durchläufe vereinheitlichen.**
- **Wir haben viele Paper gelesen und haben jede Woche eine angemessene Anzahl an Stunden mit der Projektarbeit verbracht, allerdings waren wir mit unserer Umsetzung und unseren Fortschritten nicht immer zufrieden.**

Glad

- **Wir haben tiefere Einblicke in das *Machine Learning* gesammelt, insbesondere CNNs. Zudem haben wir das Thema Remote-Sensing kennen gelernt und können somit auch mit Satelliten- oder anderen Geo-Daten arbeiten**
- **Wir haben Konzepte wie das Transfer Learning durch die praktische Umsetzung viel besser verstanden.**
- **Der zur Verfügung gestellte Code hat uns sehr beeindruckt und hilft uns bei zukünftigen Projekten dabei eine Struktur zu setzen.**

Quellen

[1] Team, K. (o. J.). Keras documentation: Transfer learning & fine-tuning. Abgerufen 15. Juli 2022, von https://keras.io/guides/transfer_learning/#:~:text=The%20most%20common%20incarnation%20of,top%20of%20the%20frozen%20layers.

[2] Pires de Lima, Rafael, und Kurt Marfurt. „Convolutional Neural Network for Remote-Sensing Scene Classification: Transfer Learning Analysis“. *Remote Sensing*, Bd. 12, Nr. 1, Dezember 2019, S. 86. *DOI.org (Crossref)*, https://doi.org/10.3390/rs12010086.

[3] How to rapidly test dozens of deep learning models in Python (https://towardsdatascience.com/how-to-rapidly-test-dozens-of-deep-learning-models-in-python-cb839b518531)