# ANN Regularization

Brandon Marquez Salazar

## I. Introduction

ANN generalization capabilities are the way an ANN behave against data was not trained with. Some of the most important thing for an ANN is the generalization capability for classification. But, when datasets are small, unbalanced or massive. Or even with large training epochs, there is a chance that the model will overfit disabling itself from generalization. A 100% of accuracy in each validation method can indicate overfitting.

Due to overfitting and low convergence fenomena, there are different methods for learning and regularization. The learning method usually enhance training time and accuracy, but in some cases, it can reach overfitting more easily; then, regularization can be applied to balance training behaviour.

## II. Core Concepts and Methods

For any big ANN with small weights, the generalization capabilities lies on the weights themselves. buth when there's a terrifying good performance at training but poor at generalization, there should be an overfitting phenomenon.

Think for a second on the classic math joke where a kid was asked about an equation, already solved days ago, to say $2y+4=0$. The kid says "I just remember the solution when it was $x$". In can be said that the kid overfitted on math class due to bad learning methods or too much equations to memorize.

Regularization is a way to let the learning model enhance generalization, affecting bias and variance of the model.

Mathematically can be defined the following way

$$\tilde{E} = E + \lambda\Omega(y)$$

Where $\Omega(y)$ is a penalty function and $\lambda$ is the regularization parameter.

### A. Tikonov Regularization

Tikonov's regularization can be described as

$$\Omega(y) = \sum_{r=0}^{R} \int_a^b h_r(x) \left(\frac{d^r y}{dx^r}\right)^2 dx$$

$$h_r(x) \geq 0 \wedge 0 \leq r \leq R \wedge h_R(x) > 0$$

The term $\Omega(y)$ should be correctly defined to help mitigate overfitting, balancing bias and variance "interaction".

### B. Pruning Regularization

The pruning refers to the removal of insignificant weights from the model, taking off nodes which are unimportant or less contributive to the model. The error function regularization plays an important role for this method.

Here the reduction can be done by

$$J = \sum_{k=1}^{N} \epsilon(i) + \lambda\epsilon_p(w)$$

where $\lambda$ is the regularization parameter.

The penalization for high weights which favours low weights comes from

$$\epsilon_p(w) = \sum_{k=1}^{K} h(w_k^2)$$

where $K$ is the number of weights and to favour weights that satisfies $|w_k| < |w_0|$ is the function $h(w_k^2)$ defined as

$$h(w_k^2) = \frac{w_k^2}{w_0^2 + w_k^2}$$

### C. $L_1$ and $L_2$ Regularization

The $L_1$ and $L_2$ regularization are the most common regularization methods for ANN.

*1) $L_1$ technique:*

$$J(w) = \sum_{i=1}^{N} e_i^2 + \lambda \sum_{i=1}^{N} ||w_i||$$

Which penalizes the high weights.

And the weight update is made by

$$\Delta w_i^{(t+1)} = -\mu \frac{\delta E(w)}{\delta w_i}\bigg|_{w_i^(t)} - \lambda\,\text{sign}(w_i^(t))$$

This solution may cause that several weights tend to be zero.

*2) $L_2$ technique:*

$$J(\mathbf{w}) = \sum_{i=1}^{N} L(f(\mathbf{x}_i), y_i) + \frac{\lambda}{2}\|\mathbf{w}\|_2^2$$

In this formulation, the $L_2$ norm of the weight vector $\mathbf{w}$ is given by $\|\mathbf{w}\|_2 = \sqrt{\sum_{j=1}^{M} w_j^2}$. Consequently, the squared $L_2$ norm simplifies to the sum of squares of all individual weights:

$$\|\mathbf{w}\|_2^2 = \sum_{j=1}^{M} w_j^2$$

*3) The Key Difference between $L_1$ and $L_2$:* While both $L_1$ and $L_2$ techniques aim to prevent overfitting by penalizing model complexity, they do so in fundamentally different ways with distinct outcomes. The core difference lies in the nature of the penalty function they employ.

$L_1$ uses an **absolute magnitude** penalty while $L_2$ uses a **squared magnitude** penalty.

### D. Restriction terms

*1) Adaptive Penalty:* The restriction term referres at this point is

$$h(w_k) = \frac{w_k^2}{w_0^2 + w_k^2}$$

*2) MaxNorm Penalty:* Thie one restricts the maximum value through a threshold for each weight.

### E. Dropout Regularization

This one is quite interesting, it focuses on the topology of the network, which varies throughout the training process providing different "subnetworks" that can achieve learning for different sets of data. This approach gets an interesting behaviour enhancing the model performance, because it can be seen as a equation system whose results affect the final output layer, where the equations that doesn't learnt some generalization just "ignores" the inputs and let's the function that actually learnt from those classes to work on it.

### F. Normalization Regularization

*1) Batch:* Here, the normalization of the output of the linear layer is made before the activation function. This can regulate the learning process, so, it's all the magic.
*2) Per instance:* Instead of batch, this normalizes by channels.
*3) Per layer:* Here, the normalization is made at each layer.

## III. Applications and Discussion

Now an experiment will be made to compare behaviour of an ANN using different regularization methods.

### A. Installation and import of modules

```
!pip install numpy
!pip install tensorflow
!pip install matplotlib
```

```
import tensorflow as tf
from tensorflow.keras import layers, models, regularizers
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, OneHotEncoder
import numpy as np
import matplotlib.pyplot as plt
```

### B. Loading the data

```
iris = load_iris()
X, y = iris.data, iris.target

# Split the data
X_train, X_test, y_train, y_test = train_test_split(
  X, y, test_size=0.2,
  random_state=42, stratify=y
)

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# One-hot encode the labels
encoder = OneHotEncoder(sparse_output=False)
y_train = encoder.fit_transform(y_train.reshape(-1, 1))
y_test = encoder.transform(y_test.reshape(-1, 1))
```

### C. Model definition

```
def create_base_model():
    model = models.Sequential([
        layers.Dense(
          64, activation='relu',
          input_shape=(4,)),
        layers.Dense(32, activation='relu'),
        layers.Dense(3, activation='softmax')
    ])
    return model
```

### D. Defining each regularization method

```
def create_l1_model(l1_lambda=0.01):
    model = models.Sequential([
        layers.Dense(
          64, activation='relu',
          input_shape=(4,),
          kernel_regularizer=regularizers.l1(l1_lambda)),
        layers.Dense(32, activation='relu',
          kernel_regularizer=regularizers.l1(l1_lambda)),
        layers.Dense(3, activation='softmax')
    ])
    return model

def create_dropout_model(dropout_rate=0.3):
    model = models.Sequential([
        layers.Dense(
          64, activation='relu',
          input_shape=(4,)
        ),
        layers.Dropout(dropout_rate),
        layers.Dense(32, activation='relu'),
        layers.Dropout(dropout_rate),
        layers.Dense(3, activation='softmax')
    ])
    return model
```

```
def create_batchnorm_model():
    model = models.Sequential([
        layers.Dense(
          64, activation='relu',
          input_shape=(4,)
        ),
        layers.BatchNormalization(),
        layers.Dense(32, activation='relu'),
        layers.BatchNormalization(),
        layers.Dense(3, activation='softmax')
    ])
    return model

models_dict = {
    'Base Model':        create_base_model(),
    'L1 Regularization': create_l1_model(l1_lambda=0.005),
    'Dropout':           create_dropout_model(dropout_rate=0.5),
    'BatchNorm':         create_batchnorm_model()
}
history_dict = {}
```

## E. Training process

```
epochs = 2000
batch_size = 128


for name, model in models_dict.items():
    print(f"\nTraining {name}...")
    model.compile(optimizer='adam',
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])

    # Train the model
    history = model.fit(X_train, y_train,
                        batch_size=batch_size,
                        epochs=epochs,
                        validation_data=(X_test, y_test),
                        verbose=0) # Set to 0 for silent training
    history_dict[name] = history
    print(f"  - Final Training Accuracy: "+
          f"{history.history['accuracy'][-1]:.4f}")
    print(f"  - Final Validation Accuracy: "+
          f"{history.history['val_accuracy'][-1]:.4f}")
```

```
Training Base Model...
  - Final Training Accuracy: 1.0000
  - Final Validation Accuracy: 0.9333

Training L1 Regularization...
  - Final Training Accuracy: 0.9833
  - Final Validation Accuracy: 1.0000

Training Dropout...
  - Final Training Accuracy: 0.9833
  - Final Validation Accuracy: 0.9667

Training BatchNorm...
  - Final Training Accuracy: 1.0000
  - Final Validation Accuracy: 0.9667
```

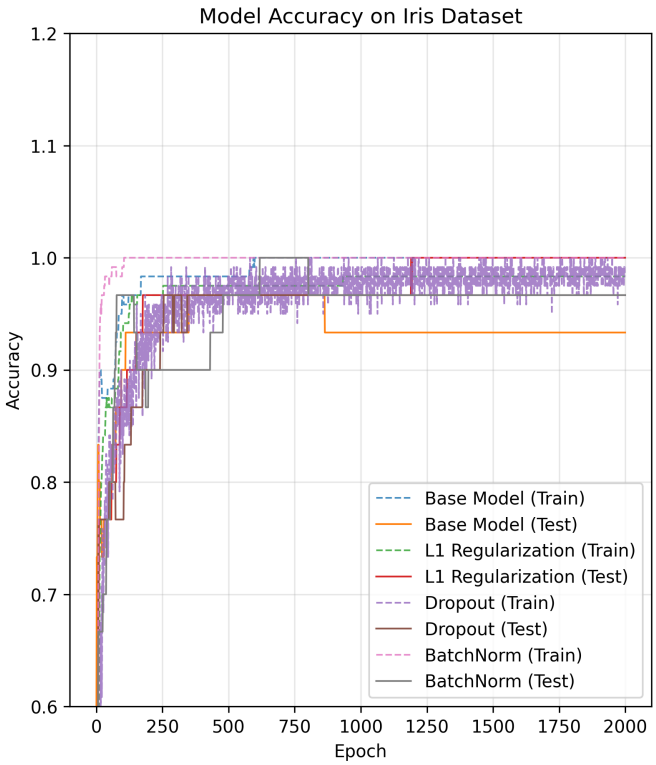
## F. Results evaluation

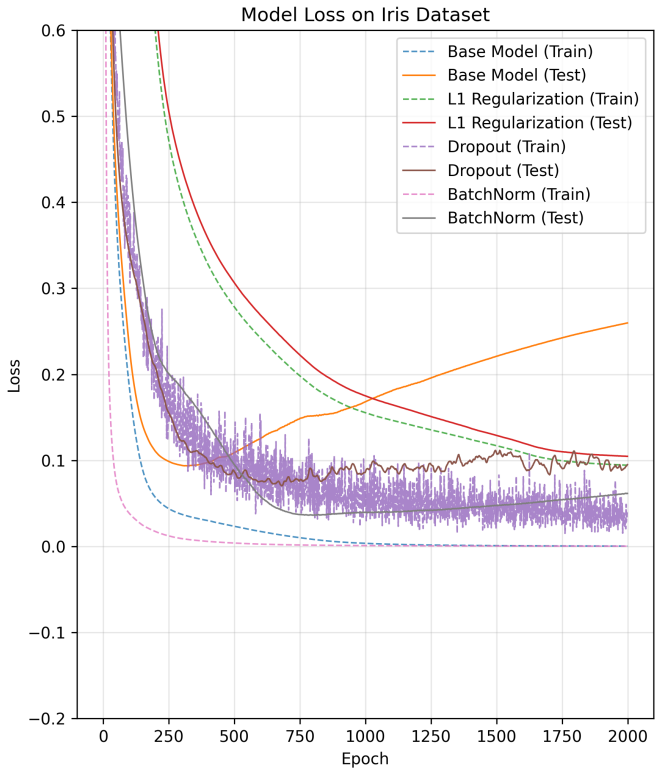
```
print("\n" + "="*50)
print("Final Test Accuracy:")
for name, model in models_dict.items():
    test_loss, test_acc = model.evaluate(X_test, y_test, verbose=0)
    print(f"{name}: {test_acc:.4f}")
```

```
==================================================
Final Test Accuracy:
Base Model: 0.9333
L1 Regularization: 1.0000
Dropout: 0.9667
BatchNorm: 0.9667
```

### 1) Training and validation accuracy:

```
plt.figure(figsize=(6, 7))
for name, history in history_dict.items():
    plt.plot(history.history['accuracy'],
             label=f'{name} (Train)',
             linestyle='--',
             alpha=0.8,
             linewidth=1
    )
    plt.plot(history.history['val_accuracy'],
             label=f'{name} (Test)',
             linewidth=1
    )
plt.title('Model Accuracy on Iris Dataset')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.ylim(0.6, 1.2)
plt.legend(loc='lower right')
plt.grid(True, alpha=0.3)
plt.show()
```




*2) Training and validation loss:*

```
plt.figure(figsize=(6, 7))
for name, history in history_dict.items():
    plt.plot(history.history['loss'],
             label=f'{name} (Train)',
             linestyle='--', alpha=0.8,
             linewidth=1
    )
    plt.plot(history.history['val_loss'],
             label=f'{name} (Test)',
             linewidth=1
    )
plt.title('Model Loss on Iris Dataset')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.ylim(-0.2, 0.6)
plt.legend(loc='upper right')
plt.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()
```

*3) Model comparison:* Different models compared by parameters and size

```
print("\nModel Complexity (Number of Trainable Parameters):")
for name, model in models_dict.items():
    total_params = model.count_params()
    print(f"{name}: {total_params:,} parameters")
```

```
Model Complexity (Number of Trainable Parameters):
Base Model: 2,499 parameters
L1 Regularization: 2,499 parameters
Dropout: 2,499 parameters
BatchNorm: 2,883 parameters
```

## IV. Conclusions

Within this experiment, we can observe the different behaviours from different regularization methods applied to a simple ANN, using the iris dataset.

The effectiveness of a regularization method usually depends on it's definition. And the most effective, leaving a good training timing without overfitting, for this particular experiment, were the Dropout and BatchNorm methods.

## References

[1] P.L. Bartlett. "The sample complexity of pattern classification with neural networks: the size of the weights is more important than the size of the network". In: *IEEE Transactions on Information Theory* 44.2 (Mar. 1998), pp. 525–536. ISSN: 0018-9448. DOI: 10.1109/18.661502. URL: http://dx.doi.org/10.1109/18.661502.

[2] Chris M. Bishop. “Training with Noise is Equivalent to Tikhonov Regularization”. In: *Neural Computation* 7.1 (Jan. 1995), pp. 108–116. ISSN: 0899-7667. DOI: 10.1162/neco.1995.7.1.108. eprint: https://direct.mit.edu/neco/article-pdf/7/1/108/812990/neco.1995.7.1.108.pdf. URL: https://doi.org/10.1162/neco.1995.7.1.108.

[3] Babak Hassibi and David G. Stork. “Second order derivatives for network pruning: optimal brain surgeon”. In: *Proceedings of the 6th International Conference on Neural Information Processing Systems*. NIPS’92. Denver, Colorado: Morgan Kaufmann Publishers Inc., 1992, pp. 164–171. ISBN: 1558602747.

[4] Sergey Ioffe and Christian Szegedy. “Batch normalization: accelerating deep network training by reducing internal covariate shift”. In: *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*. ICML’15. Lille, France: JMLR.org, 2015, pp. 448–456.

[5] 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.