IT496: Introduction to Data Mining

Lecture 29-30

Overfitting in Neural Networks

(Slides are created from the lecture notes of Dr. Derek Bridge, UCC, Ireland)

Arpit Rana

26th - 27th October 2023

Problems and Solutions in Training Deep Neural Network

- *Vanishing Gradient Problem* ✔
  - Better Initialization
  - Non-saturating Activation Functions
  - Batch Normalization
- *Need enough (labelled) training data*
  - Reusing pretrained layers (transfer learning)
  - Unsupervised pre-training
  - Pre-training on an auxiliary task
- *Training may be extremely slow*
  - Using faster optimizers
  - Learning rate scheduling
- *Overfitting*
  - Reducing the network size
  - Weight/Max-norm regularization
  - Dropout
  - Early stopping

We will not cover these topics in detail.

Overfitting

DNNs typically have tens of thousands of parameters, sometimes even millions.

- This increases their (model) capacity: ability to fit a huge variety of complex datasets.
- This also makes the network prone to overfitting the training set.

Overfitting

Reminder. If your model overfits, your main options are:

- gather more training examples;
- remove noise in the training examples;
- change model: move to a less complex model;
- simplify by reducing the number of features;
- stick with your existing model but add constraints (if you can) to reduce its complexity.

Overfitting in Neural Networks

We will look at the following wayouts.

- **reducing the network's size** — an example of moving to a less complex model;
- **weight regularization** — an example of adding constraints to reduce complexity;
- **dropout** — an example of adding constraints to reduce complexity;
- **max-norm regularization** – again an example of adding constraints to reduce complexity; and
- **early stopping** — a somewhat different way of avoiding overfitting.

A Network that Overfits a Little

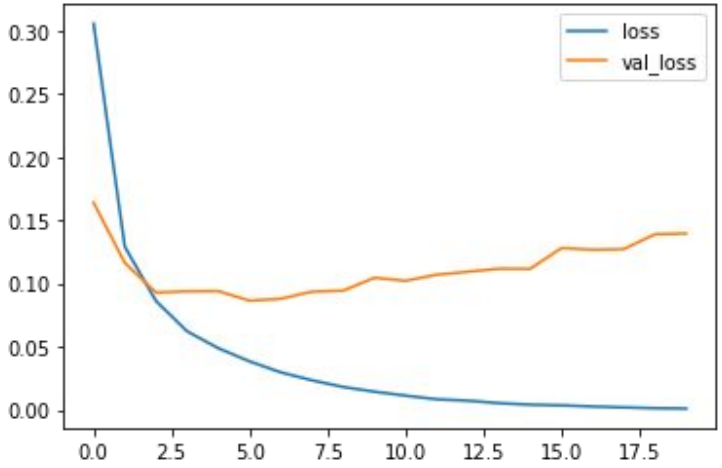
```
# A network that overfits (a little!)

inputs = Input(shape=(28 * 28,))
x = Rescaling(scale=1./255)(inputs)
x = Dense(1024, activation="relu")(x)
x = Dense(1024, activation="relu")(x)
outputs = Dense(10, activation="softmax")(x)
overfitting_model = Model(inputs, outputs)
overfitting_model.compile(optimizer=RMSprop(learning_rate=0.0001), loss="sparse_categorical_crossentropy")
```

```
history = overfitting_model.fit(mnist_x_train, mnist_y_train, epochs=20, batch_size=32,
                                verbose=0, validation_split=0.20)
history.history["loss"][-1], history.history["val_loss"][-1]
```

(0.0010047738905996084, 0.13949735462665558)

A Network that Overfits a Little

```
pd.DataFrame(history.history).plot()
```



Reducing Network Size

- We can make the model (neural network) less complex by reducing the number of parameters.
- Obviously enough, this is achieved by:
  - reducing the number of hidden layers, and/or
  - reducing the number of neurons within the hidden layers.

Reducing Network Size

```
# Smaller network

inputs = Input(shape=(28 * 28,))
x = Rescaling(scale=1./255)(inputs)
x = Dense(256, activation="relu")(x)
outputs = Dense(10, activation="softmax")(x)
smaller_model = Model(inputs, outputs)
smaller_model.compile(optimizer=RMSprop(learning_rate=0.0001), loss="sparse_categorical_crossentropy")
```

```
history = smaller_model.fit(mnist_x_train, mnist_y_train, epochs=20, batch_size=32, verbose=0, validation_split=0.2)
history.history["loss"][-1], history.history["val_loss"][-1]
```

(0.06080113351345062, 0.09995315223932266)

Weight Regularization

- For linear regression, we used regularization to ensure that the coefficients β took only small values by penalizing large values in the loss function.
    - **Ridge**: we penalized by the l_2-norm (the sum of their squares).
    - **Lasso**: we penalized by the l_1-norm (the sum of their absolute values).
    - **Elastic Net**: we penalized by the mix of both of the above.
    - A **hyperparameter** λ, called the 'regularization parameter' controlled the balance between fitting the data versus shrinking the parameters.

Weight Regularization

- Weight Regularization in neural networks is the same idea, but applied to the weights in the layers (not their biases) of a network.

```
layer = keras.layers.Dense(100, activation="elu",
                           kernel_initializer="he_normal",
                           kernel_regularizer=keras.regularizers.l2(0.01))
```

- The `l2()` function returns a regularizer that will be called to compute the regularization loss, at each step during training. This regularization loss is then added to the final loss.

- You can use `keras.regularizers.l1()` for l_1 regularization and `keras.regularizers.l1_l2()` for both l_1 and l_2 regularization.

Weight Regularization

- In case of applying the same activation function, initialization strategy, and the same regularizer to all layers, you can use Python's `functools.partial()` function.
- It lets you create a thin wrapper for any callable, with some default argument values.

```
from functools import partial

RegularizedDense = partial(keras.layers.Dense,
                           activation="elu",
                           kernel_initializer="he_normal",
                           kernel_regularizer=keras.regularizers.l2(0.01))

model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    RegularizedDense(300),
    RegularizedDense(100),
    RegularizedDense(10, activation="softmax",
                     kernel_initializer="glorot_uniform")
])
```

Weight Regularization

```
# Regularized network

inputs = Input(shape=(28 * 28,))
x = Rescaling(scale=1./255)(inputs)
x = Dense(1024, activation="relu", kernel_regularizer=l2(0.0001))(x)
x = Dense(1024, activation="relu", kernel_regularizer=l2(0.0001))(x)
outputs = Dense(10, activation="softmax", kernel_regularizer=l2(0.0001))(x)
regularized_model = Model(inputs, outputs)
regularized_model.compile(optimizer=RMSprop(learning_rate=0.0001), loss="sparse_categorical_crossentropy")
```

```
history = regularized_model.fit(mnist_x_train, mnist_y_train, epochs=20, batch_size=32, verbose=0, validation_split=0.2)
history.history["loss"][-1], history.history["val_loss"][-1]
```

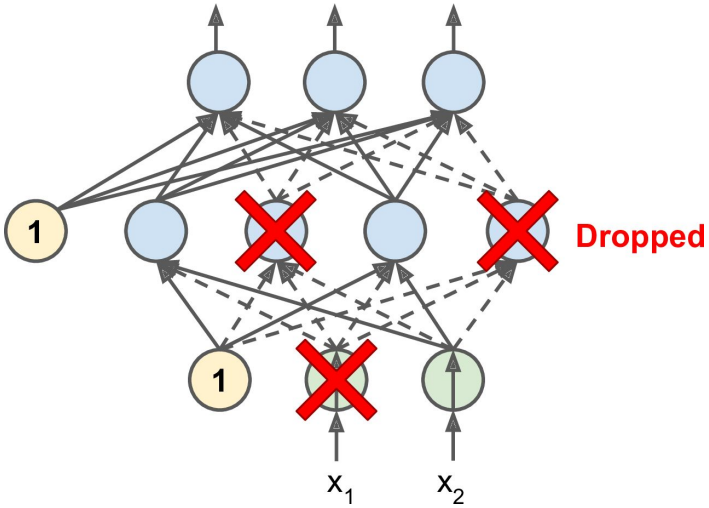
(0.056435953825712204, 0.11559036374092102)

- Weight regularization can work well when the network is small but has little effect on larger networks.
- For larger networks, a better option can be ***dropout***.

Dropout

Imagine we have a layer that uses dropout with dropout rate ***p***, e.g., ***p*** = 0.5

Then, in a given step of the backprop algorithm, each neuron in the layer (any layer except output layer) has probability ***p*** of being ignored — treated as if it were not there.



One Way of Doing Dropout

- **Training**. For any given mini-batch: // dropout rate is ***p***.
    - In the forward propagation,
        - decide which neurons will be dropped (chosen with probability ***p***);
        - set the activations of the dropped neurons to zero;
        - multiply the activations of the kept neurons by 1/(1 - ***p***).
    - In the backpropagation, ignore the dropped out neurons.

  Note that different neurons will get dropped for each mini-batch.

- **Testing**. The `Dropout` layer does nothing at all, it just passes the inputs to the next layer.

One Way of Doing Dropout

- But why did we multiply activations by 1/(1 - ***p***) ?
    - In testing, for ***p*** = 0.5 a neuron in the next layer will receive input from on average twice as many neurons as it did in training.
    - The multiplication by 1/(1 - ***p***) compensates for this. This is called *keep probability*.

Why Does Dropout Reduce Overfitting?

- Consider a company whose employees were told to toss a coin every morning to decide whether to go to work or not.
    - The organization wants its employees must become more like generalists, less like specialists.
    - The organization would need to become more resilient. It could not rely on any one employee to perform critical tasks: the expertise would need to be spread across many employees.

  Similarly, in dropout layers, neurons learn more robust features.

Why Does Dropout Reduce Overfitting?

- Another way to think about it.
    - Since a neuron can be present or absent, it's like training on a different neural network at each step.
        - There is a total of 2^N possible networks (where N is the total number of droppable neurons).
    - The final result is a bit like an ensemble of these many different virtual neural networks.

However, it typically increases the number of epochs needed for convergence (roughly double when $\boldsymbol{p} = 0.5$).

Dropout in Keras

```
# Network with dropout

inputs = Input(shape=(28 * 28,))
x = Rescaling(scale=1./255)(inputs)
x = Dense(1024, activation="relu")(x)
x = Dropout(0.5)(x)
x = Dense(1024, activation="relu")(x)
x = Dropout(0.5)(x)
outputs = Dense(10, activation="softmax")(x)
model_with_dropout = Model(inputs, outputs)
model_with_dropout.compile(optimizer=RMSprop(learning_rate=0.0001), loss="sparse_categorical_crossentropy")
```

```
history = model_with_dropout.fit(mnist_x_train, mnist_y_train, epochs=20, batch_size=32, verbose=0, validation_split=0.2)
history.history["loss"][-1], history.history["val_loss"][-1]
```

(0.05647280439734459, 0.11740138381719589)

Monte Carlo (MC) Dropout

Yarin Gal and Zoubin Ghahramani (2016) introduced a powerful technique called *MC Dropout*:

- MC Dropout is the *use of dropout at inference time (forward passes only)* in order to add stochasticity to a network that can be used to generate a cohort of predictors/predictions to perform statistical analysis.
    - This can boost the performance of any trained dropout model, without having to retrain it or even modify it at all!
    - It also provides a much better measure of the model's uncertainty, and
    - It is also amazingly simple to implement.

Monte Carlo (MC) Dropout

Look at the following code that fully implements the MC *dropout*:

```
y_probas = np.stack([model(X_test_scaled, training=True)
                     for sample in range(100)])
y_proba = y_probas.mean(axis=0)
```

- The `model(X)` is similar to `model.predict(X)` except it returns a *tensor* rather than a Numpy array and it supports the `training` argument.
- So, we make 100 predictions over the test set and we stack them. As the dropout is on (`training=true`), all predictions will be different.
- The shape of `y_probas` will be [100, 10000, 10]. We average over the first dimension (`axis=0`), and get `y_proba`, an array of shape [10000, 10].

Monte Carlo (MC) Dropout

- Let's look at the model's prediction for the first instance in the test set, with dropout off:

```
>>> np.round(model.predict(X_test_scaled[:1]), 2)
array([[0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.01, 0.  , 0.99]],
      dtype=float32)
```

- Compare this with the predictions made when dropout is activated:

```
>>> np.round(y_probas[:, :1], 2)
array([[[0.  , 0.  , 0.  , 0.  , 0.  , 0.14, 0.  , 0.17, 0.  , 0.68]],
       [[0.  , 0.  , 0.  , 0.  , 0.  , 0.16, 0.  , 0.2 , 0.  , 0.64]],
       [[0.  , 0.  , 0.  , 0.  , 0.  , 0.02, 0.  , 0.01, 0.  , 0.97]],
       [...]
>>> np.round(y_proba[:1], 2)
array([[0.  , 0.  , 0.  , 0.  , 0.  , 0.22, 0.  , 0.16, 0.  , 0.62]],
      dtype=float32)
```

- Looking at the standard deviation of the probability estimates.

```
>>> y_std = y_probas.std(axis=0)
>>> np.round(y_std[:1], 2)
array([[0.  , 0.  , 0.  , 0.  , 0.  , 0.28, 0.  , 0.21, 0.02, 0.32]],
      dtype=float32)
```

Monte Carlo (MC) Dropout

If our model contains special layers during training (e.g., `BatchNormalization`), we should not force training mode as earlier.

- Instead, we create the subclass of the `Dropout` layer and override its `call()` method to force its `training` argument to `True`.

```
class MCDropout(keras.layers.Dropout):
    def call(self, inputs):
        return super().call(inputs, training=True)
```

- We then use our new `MCDropout` layer in place of `Dropout` when defining our Sequential model.

Note: The number of Monte Carlo samples that we use (e.g., 100 in our previous example) is a hyperparameter: higher value gives more accurate predictions, however, increases the inference time.

Max-Norm Regularization

- For each neuron, it constrains the weights w of the incoming connections such that $\|w\|_2 \leq r$, where r is the max-norm hyperparameter and $\|\cdot\|_2$ is the l_2 norm.
- It does not add a regularization loss term to the overall loss function.
    - Instead, it is typically implemented by computing $\|w\|_2$ after each training step and rescaling w if needed,

$$w = \frac{wr}{\|w\|_2}$$

- Reducing r increases the amount of regularization and helps reduce overfitting. It can also help alleviate the unstable gradients (in absence of Batch Normalization).

Max-Norm Regularization

- In Keras, set every hidden layer's `kernel_constraint` argument to a `max_norm()` constraint, with the appropriate max value,

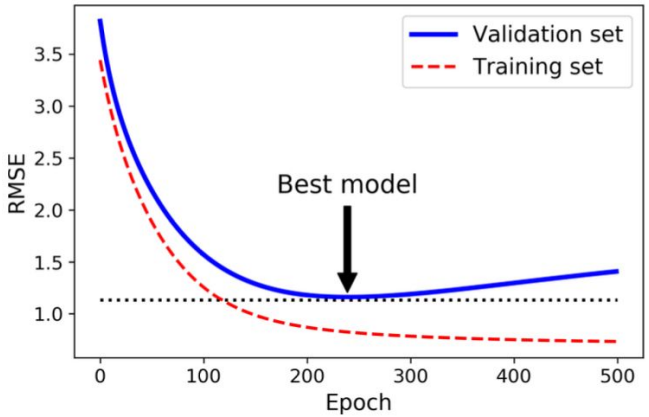
```
keras.layers.Dense(100, activation="elu", kernel_initializer="he_normal",
                   kernel_constraint=keras.constraints.max_norm(1.))
```

- After each training step, the model's `fit()` method will call the object returned by `max_norm()`, passing it the layer's weights and getting rescaled weights in return, which then replace the layer's weights.

- We can also constrain the bias terms by setting the `bias_constraint` argument.

Early Stopping

- We know that a sign of overfitting is that validation error stops getting lower and starts getting larger.
- We can exploit this during Gradient Descent as another way of avoiding overfitting, known as early stopping:
  - During Gradient Descent, monitor validation error (or loss).
  - Interrupt training when the validation error has stopped improving for a certain number of epochs.



Early Stopping

In Keras, the `fit()` method accepts a `callbacks` argument that lets us specify a list of objects that Keras will call -

- at the start and end of training,
- at the start and end of each epoch, and even
- at the start and end of processing each batch.

Early Stopping in Keras

- In Keras, this is done using the `EarlyStopping` callback.
- The `patience` argument allows you to specify how many epochs must pass with no improvement relative to the current best.
- `restore_best_weights=True` restores the weights and biases from when validation error was at its lowest.

```
history = overfitting_model.fit(mnist_x_train, mnist_y_train, epochs=20, batch_size=32,
                                verbose=0, validation_split=0.2,
                                callbacks=[EarlyStopping(monitor="val_loss", patience=2, restore_best_weights=True)])
```

```
[(l, v) for l, v in zip(history.history["loss"], history.history["val_loss"])]
```

```
[(0.0012100160820409656, 0.1484818160533905),
 (0.0009289713925682008, 0.1520228236913681),
 (0.0004663218569476157 4, 0.15337863564491272)]
```

Early Stopping in Keras

- An advantage of early stopping is that we can be less concerned about choosing the number of epochs: just use something very large.

- But, now we have the problem of deciding on the `patience`. If runtime is your problem, then you can choose a low value. Otherwise, you choose a low value for 'easier' problems!

Conclusions

- Overfitting is a major problem but has many solutions.
- There are lots of solutions in addition to the ones above:
  - Remember **Batch Normalization** has a regularizing effect.
  - There are other techniques that we won't cover (e.g. Gradient clipping).
  - There are the things we've mentioned in an earlier lecture, especially getting more data!

Next lecture

Convolutional Neural Networks (CNNs)