Universidade de São Paulo – USP São Carlos
Instituto de Ciências Matemáticas e de Computação
SCC5809 – Redes Neurais – Profa. Dra. Roseli Romero
Damares Oliveira de Resende – #11022990

Relatório de Atividades – Projeto II

Nesse exercício criou-se uma rede neural convolucional utilizando os frameworks TensorFlow e Keras para a construção e treinamento do modelo. A finalidade desse modelo é a de classificar 8 expressões faciais distintas, a saber: raiva, nojo, medo, feliz, negativa, neutra, triste e surpresa.

A base de dados usada, chamada QIDER, possui 4521 imagens de treinamento e 1346 imagens de validação. Além destas imagens, 8 imagens novas (uma de cada classe) foram utilizadas para testar a performance do modelo gerado. Essas novas imagens foram disponibilizadas juntamente com o código. Elas estão em RGB porém a função *tf.image.decode_jpeg* está configurada para automaticamente converter as imagens carregadas para a escala de cinza se necessário.

Pre-processamento das imagens

As imagens da base de dados QIDER, além de estarem consideravelmente desbalanceadas, possuem também tamanhos diferentes. O primeiro passo do pré-processamento consiste em redimensionar essas imagens para um tamanho padrão. Testou-se a princípio a função *resize_image_with_crop_or_pad,* que corta imagens grandes e coloca um *padding* em imagens pequenas. Porém, como o tamanho das imagens varia bastante, esse método foi substituído pela função *resize_images.* Todas as imagens foram redimensionadas para o tamanho 96x96.

O próximo passo do pré-processamento consiste em balancear as classes. Para isso, fora criado o método *augment_dataset,* que baseado na distribuição das classes gera novas imagens. As imagens geradas consistem na rotação de cada imagem em 180˚ ao longo do eixo Y, o que dobra o tamanho do dataset, e posteriormente na adição de um ruído gaussiano nelas, quadruplicando assim o volume de dados. A Tabela 1 mostra a distribuição de classes para a base de treinamento antes e depois do processo de expansão.

	C1	C2	C3	C4	C5	C6	C7	C8
A	501	377	336	746	1618	115	424	404
D	1618	1508	1344	1618	1618	460	1618	1616

Tabela 1: Distribuição das classes antes (A) e depois (D) da expansão da base de dados. C1, C2, C3, C4, C5, C6, C7 e C8 correspondem respectivamente às classes: raiva, nojo, medo, feliz, negativa, neutra, triste e surpresa.

Mesmo após o processo de expansão, a classe C6, referente a expressões neutras, continua desbalanceada. Para equilibrar isso, implementou-se o método *_class_weights*, o qual calcula um peso para cada classe baseado em sua distribuição. Esse peso é aplicado durante o treinamento do modelo na camada *softmax*, alterando assim a probabilidade de cada classe e evitando que C6 fique muito favorecida em relação às outras. O peso das classes é calculado pela Equação 1:

$$2 - \frac{nc - min}{\max - min}$$

Equação 1: Calcula o peso de cada classe, onde *nc* corresponde ao número de exemplos da classe *c*, *min* corresponde ao número mínimo de exemplos considerando todas as classes, e *max* corresponde ao número máximo.

A Equação 1 é baseada no processo de normalização, definido pela fração na equação e aqui chamado *norm*. A ideia é normalizar o vetor que indica a distribuição de classes. Isso fará com que cada valor fique entre 0 e 1, onde zero é a classe com menos amostras e 1 a classe com mais amostras. Porém, como a classe com menos amostras deve ter um peso maior e a com mais amostras deve ter um peso menor, foi necessário inverter a lógica da normalização. Por isso o 1 – *norm* na equação. Contudo, o peso de uma classe deve ser no mínimo 1. Logo deve-se adicionar 1 ao resultado, e por isso a Equação 1 se reduz à 2 – *norm*.

Organização do código

O código está dividido em 5 scripts diferentes: *cnn_run.py, cnn_eval.py, cnn_data.py, cnn_model.py e cnn_params.py*. Escolheu-se fazer isso para modularizar cada parte do processo de treinamento e avaliação do modelo.

O script *cnn_params.py* apenas define os parâmetros do algoritmo, tais como o tamanho das imagens, o número de épocas e as camadas da CNN. *cnn_data.py* é responsável por ler as imagens, processá-las e criar os *datasets* de treinamento, validação e teste. *cnn_model.py* define o modelo. Esse script possui métodos para construir a CNN, treina-la, avalia-la e salva-la. *cnn_run.py* é o script principal. Ele carrega os dados, constrói e treina o modelo, e gera os gráficos de resultado. Por fim, *cnn_eval.py* avalia o modelo na base de testes criada. Ele carrega o modelo treinado e salvo por *cnn_run.py* e avalia o desempenho do mesmo nas novas imagens.

Arquitetura da CNN

A arquitetura da CNN é definida no script *cnn_params.py.* A rede recebe como entrada uma imagem 96x96x1 em batches de tamanho 32. A Tabela 2 mostra os parâmetros usados em cada camada. Para todas as camadas foi usada a função ReLU como função de ativação e para a camada de saída utilizou-se a função *softmax*.

Camada	#Units	Kernel	Max Pool	Dropout
Conv1	32	3x3	2x2	-
Conv2	16	3x3	-	0.3
Conv3	24	3x3	2x2	0.1
Densa1	24	-	-	0.4
Densa2	16	-	-	0.1
Densa3	8	-	-	-

Tabela 2: Arquitetura da CNN. Onde #Units indica o número de unidades. Para as camadas convolucionais ele corresponde aos filtros; para as camadas densas ele corresponde ao número de neurônios. Max Pool corresponde à camada posterior à camada convolucional. Dropout indica a porcentagem de dropout em cada camada. " - " indica ausência de uma determinada camada ou etapa.

Resultados e conclusão

Várias arquiteturas de rede foram testadas ao longo deste trabalho, incluindo diferentes números e tamanho de camadas, aplicação ou não de *dropout* e *max_pooling*, número de épocas, tamanho do batch, tamanho das imagens e balanceamento ou não da base de dados. Neste relatório serão mostrados os resultados do modelo indicado na Tabela 2 com o balanceamento de classes explicado anteriormente.

A partir do modelo descrito na Tabela 2, duas variações foram testadas - com e sem *dropout*. A Figura 1 mostra os resultados da acurácia e erro no treino e teste do modelo sem *dropout*. A Figuras 2 mostra o mesmo para o modelo com *dropout*.

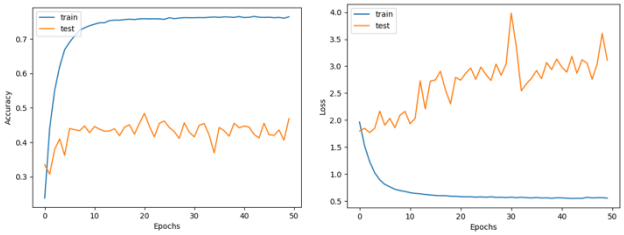

Figura 1: Acurácia (gráfico à esquerda) e perda (gráfico à direita) do modelo SEM *dropout* ao longo de 50 épocas.

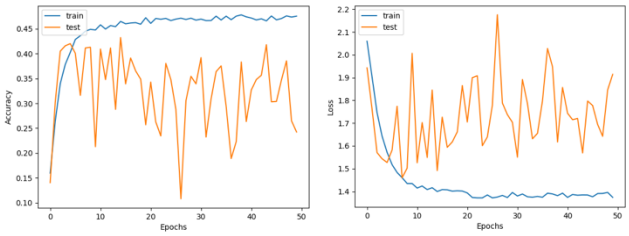

Figura 2: Acurácia (gráfico à esquerda) e perda (gráfico à direita) do modelo COM *dropout* ao longo de 50 épocas.

A Tabela 3 sumariza o resultado da acurácia e da perda para ambos os modelos nas bases de treino, validação e teste. Onde a base de validação corresponde à base disponibilizada para validação, e a base de treino é composta pelas 8 imagens geradas.

	SEM *Dropout*			**COM *Dropout***		
	Treino	Valid	Teste	Treino	Valid	Teste
ACC	0.764	0.469	0.250	0.475	0.242	0.375
LOSS	0.553	3.109	9.274	1.374	1.913	2.557

A partir da análise dos gráficos da Figura 1, observa-se que durante o treinamento a CNN está aprendendo, pois sua acurácia aumenta e seu erro diminui. Porém, esse aumento ou declínio se dá de forma exponencial, o que indica um *overfitting* no modelo. Isso pode ser confirmado ao se observar as curvas da validação. Apesar da acurácia em treinamento estar aumentando e o erro diminuindo, a acurácia em teste varia em torno de 45% e o erro aumenta a cada iteração. Conclui-se então que a CNN está memorizando a classificação.

Para evitar a memorização, foram acrescentados os *dropouts* em algumas camadas. O resultado pode ser observado na Figura 2. Verificando as curvas de teste, entende-se que as falhas de classificação estão mais coerentes com o treinamento. Porém a classificação ainda é muito instável. Além disso, a acurácia do modelo treinado caiu de cerca de 70% para 50%. Este modelo não pode ser considerado melhor que o primeiro modelo, mas pode ser considerado mais generalista.

A partir da análise da Tabela 3 observa-se que no conjunto de teste o modelo com *dropout* obteve uma performance superior ao primeiro modelo. O que reafirma a hipótese que o segundo modelo é mais generalista. Porém nenhum deles possui uma performance satisfatória, apesar de para todos os casos eles serem melhores que uma classificação aleatória.